Validation and Verification of Software Design Using Finite State Process

by

Simon C. Stanton, B. Sc.

A dissertation submitted to the School of Computing in partial fulfilment of the requirements for the degree of Bachelor of Science with Honours

University of Tasmania

November 2002

I, Simon Charles Stanton, declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any tertiary institution. To my knowledge and belief, this thesis contains no materials previously published or written by another person except where due reference is made in the text of the thesis.

Simon Charles Stanton

Abstract

This thesis aims to evaluate the effectiveness, at eliminating errors from a design specification, of a formal language (Finite State Process) automated verification tool (Labelled Transition System Analyser). The language FSP is used to model the problem domain (a version of the Lift Problem), based on a provided specification that was refined with a validation-led methodology. The validation-led model is translated (mapped) to a finite state domain wherein we test this new model for errors in the translation, for errors in the understanding of the initial requirements, and for faults in the concurrency properties of the identified co-operating entities. Exposition of errors drives their resolution. The resolution of errors gives rise to an evolutionary corrected model. The corrected model is then used as a specification for input to Implementation phases of software engineering, or, the corrected model may be used as input back to the client as text descriptions. Input returning to the client validates the problem solution, enabling a new cycle of modelling and design. Performing a documented process of FSP modelling over the evolutionary descriptions brings to light some issues that suggest the inclusion of formal methods in the design process has value due to the early removal of errors. Specifically, fatal errors, non-fatal errors and contributory specifications have all been explicitly realised from the FSP process - allowing the assertion that the formal method examined in this thesis assists the software engineering process.

Acknowledgements

Vishv Malhotra, for his support, instruction and guidance throughout the year,

Phaedra, for living with a thesis,

Julian Dermoudy for his help and encouragement,

All the honours students, for being there,

Debbie Ploughman for proofreading,

Ian Cumming, Frank Sainsbury and Jonathan Elliott for their help and assistance with blizzard, and

TPAC (Tasmanian Partnership for Advanced Computing) for access to blizzard, SGI Origin 3400.

Table of Contents

List of Figures

List of Tables

1 Introduction

Software Engineering is a complex task, the more so the larger the scale of the task, and the intricacy of the detail that the project demands. Development of software moves through different stages of evolution, commonly referred to as the Software Development Lifecycle (Section 2.3).

Each stage of the Software Development Lifecycle has a vulnerability to the introduction of errors as well as a susceptibility to the amplification of errors that are present from a previous stage.

This thesis aims to examine the effect on the resolution of errors in the design stages of software development using a verification tool. The documentation from the verification process will be weighed for its contribution to the software specification.

1.1 Software Engineering Models

It is arguably the case that models are the central abstraction of software development. Models can be loose descriptions or precise specifications, the key commonality is that models lead to an implementation, and the implementation itself is a model represented in a precise syntactic form.

As models develop they accrue greater meaning, a weight of abstractions. Therefore, communicating the full intent and the total meaning of models is deeply complicated by the need to translate between models. Translation between models is also complicated because not all models have a precise formal syntax, let alone precise formal semantics.

1.2 Error Generation Phases

These difficulties offer the software development effort a mine for defect production without having even examined the *problem space* itself for internal consistencies, or conversely, ambiguities. Errors in the development of software

can have far-reaching consequences. Somerville reports that the cost of changing a system is large and larger so the later in the software development lifecycle the changes are made (1995 p70).

The principal vulnerabilities to error generation that arise during the design stages are in the formation of the initial descriptions of the problem (Section 3.1.1), in the evolution of the developer model (Section 3.1.2) and in the output production of the design specifications for Implementation. The last category refers to a set of documents that will hereinafter be referred to as the Implementation Specification.

1.3 Validation-led Development

Validation-led development seeks to redress the generation of errors by "analysing and developing the system specifications (to lead) to a complete and consistent specification" (Lakos & Malhotra, 2002 p58). Particularly, Validation-Led Development is targeted at integrating specifications of software with validation of the software. This approach serves both the client, through the validation of their requirements, and the developer, through validation of the specification against client requirements. Ultimately, the client and the developer are served by the scrutiny applied to the initial design and modelling stages of the development effort (p57).

1.4 Validation of Software Design using Finite State Process

The formal language Finite State Process (Section 2.1.1) allows a model representation of co-operating objects that can be verified with the aid of an automated tool - the Labelled Transition System Analyser (LTSA, presented in Section 2.1.2). Application of formal concurrency analysis principally seeks to redress the generation of errors that are the result of inconsistencies in the model.

By automating the state-space exploration of behavioural interactions, verification has the potential to contribute to incrementally-improved models. Such models

have an increasing level of accuracy that is useful for updating the original problem domain, and thereby, the initial description of the problem domain as provided by the client.

1.4.1 Case Study: The Lift Problem

The effect on the software design process of the inclusion of a formal analysis stage into the validation-led process should be evaluated in the context of the development of the *design implementation* of the Lift Problem description. The Lift Problem description in this thesis is derived from the Lift Problem description and models utilised by Lakos & Malhotra in Validation-Led Development (2002, p58).

The Lift Problem offers a deceptively simple initial specification in tandem with discrete, co-operating but independent objects that have dissimilar, but dependent, lifecycles. Those objects are the Passengers and the Lift. The derivation of the Lift Problem description that is used in this thesis is presented in Section 4.2, the original (Lakos & Malhotra) Lift Problem description is presented in Section 11.1. Section 6.3 discusses the derivation of the version used in this thesis.

1.4.2 Implementation Specification

The research process is to perform FSP (Finite State Process) analysis on models of the Lift Problem - in an iterative fashion; such that the results of the analysis will assist in the redefinition of the original models as well as the analysis model.

As iterations progress, a refined, verified model will be produced. This model is here termed the ‘corrected model’ (Section 3.4.6). The corrected, or verified, model will then be translated to an Implementation Specification (Section 3.4.3) which will in this thesis be comprised of an UML object model and a set of UML behavioural diagrams.

1.5 Overview of Thesis

Section 2 Theory Review and Current Work gathers theory from fields that inform this thesis with an emphasis on formal component languages, and also presents some examples of the tools that are to be used.

Section 3 Research Question is a theoretical posing of the question of this thesis, which leads into the specification of the practical task itself in *Section 4 Problem Statement*.

The methods and approaches used to complete the practical task are detailed in *Section 5 Research Process* which also presents the documentation and deliverables of the thesis. *Section 6 Implementation Details* consolidates disparate issues of the Research Process that have nevertheless had influence upon the thesis.

Section 7 Research Discussion recaps the methodology as applied in the thesis and presents issues that have had direct influence upon the process.

Section 8 Conclusion summarises the assessment of the contribution of the FSP-modelling process to the solution model.

Section 9 Further Work suggests specific activities in respect to the final solution model presented in this thesis, as well as activities for utilising the FSP methodology elsewhere. *Section 10 References* is followed by the documentation and full FSP model listings in *Section 11 Appendix*.

2 Theory Review and Current Work

This section presents the formal FSP language and the associated checking tool, the Labelled Transition System Analyser (LTSA). The relationship of LTSA and FSP to the Darwin Architectural Modelling Language is described in the context of the emerging field of Software Architecture. These accounts are placed against the background of software development lifecycles and formal verification methods. This section concludes with an examination of the Validation-Led Development methodology.

2.1 Finite State Process and the Labelled Transition System Analyser

The Labelled Transition System Analyser (LTSA) is Java software that is available online (http://www-dse.doc.ic.ac.uk/concurrency/) or, alternatively, as an accompaniment to the Wiley & Sons publication *Concurrency: State Models & Java Programs* (Magee & Kramer, 1999). The authors, Professors Jeff Magee and Jeff Kramer, are from the Distributed Software Engineering Group at the Department of Computing, Imperial College of Science and Technology and Medicine at the University of London.

Magee and Kramer define a process as 'the execution of a sequential program', that 'a process ... has state, modified by indivisible or atomic actions' (p11). Thus, in transforming state a process engages in an action that has an effect upon the variables of the program. This is the basis for their representing 'processes as finite state machines' (p11). Magee writes that the LTSA tool 'is being developed to meet the needs for a ... tool targeted at behavioural analysis of software architectures (as) described in the Darwin architectural description language' (1999, p634).

2.1.1 Finite State Process

FSP is a process calculus notation that draws from Milner's *Calculus of Communicating Systems* (1980) and from Hoare's *Communicating Sequential Processes* (1985), as is explained by Magee and Kramer (1999: p7).

Magee and Kramer advocate the modelling of system components as atomic processes. Components that can be modelled as composites of components should be decomposed so that systems can ultimately be regarded as a composite of indivisible processes. The arrangement of member components forms the system architecture. The composition of processes allows interaction and behaviour to be defined.

To present a simple example of this approach, the following Finite State Process representation of a *coin toss* is reproduced from *Concurrency: State Models and Java Programs* (1999, p17):

```
COIN     =     (      toss -> heads -> COIN
               |      toss -> tails -> COIN
               ).
```

Modelled as a single process, COIN can engage in an action (toss) that can have either one of two results - heads or tails. This represents a non-deterministic choice, as the process may engage in either action, which leads to a change in state (modelled as an action) which leads to a return to the stable COIN state (the process is not engaged in a transformation between states).

Fine variation of loads and conditions can be modelled in FSP. FSP incorporates a mechanism for adjusting priority settings for actions and properties so as to be able to model both fair and unfair situations. The Readers and Writers problem, when modelled in FSP (Magee & Kramer, 1999 p144), allows the increasing or decreasing of priority for readers or writers, thus enabling the simulation of the different conditions that may be found in a real-world system.

2.1.2 Labelled Transition System Analyser

LTSA enables the machine verification, or automated checking, of models that are expressed in the Finite State Process (FSP) process-calculus. The LTSA tool parses an FSP description such as the one given for COIN, above, and can then

produce a graphical representation of the state space that this process maximally occupies. The state space is explored using compositional reachability analysis (Giannakopoulou, Magee & Kramer, 1999, p1).

LTSA can scale to accommodate models with large numbers of states. Magee refers to the 'Active Badge System' in Behavioural Analysis of Software Architectures using LTSA which extended to '566,820 reachable states and 2,428,488 transitions' (1999, p637).

Sample graphical LTSA output is reproduced in *Figure 1 LTSA representation of the COIN FSP Process*.

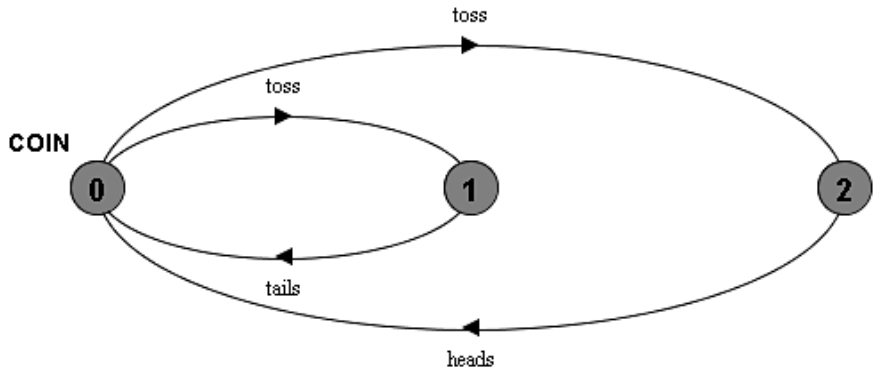

Figure 1 LTSA representation of the COIN FSP Process

Transition from state 0 to state 1 or 2 is achieved by following the non-deterministic toss action, transit back to the initial state informs the process of the outcome of the action – either heads or tails.

This simple example demonstrates how the process calculus is represented in LTSA. It is evident that LTSA can perform thorough and rigorous exploration of state space. There is no difficulty in enumerating the state space for COIN, even without the aid of a software tool. For processes of greater complexity, machine verification becomes vital.

2.1.3 FSP Modelling of Concurrent Processes

Magee et al (2000, p499) discuss the issue of communicating architectural issues and concurrency concerns with end-users and stakeholders of the system that is being designed. They express difficulty in expressing the clear meanings of property violation and deadlock traces when these issues are in a formal notation.

The clarity of understanding is considerably improved when the same information is demonstrated as an animation in the LTSA tool. The visual appreciation of a path to deadlock (for example) is appreciable at a conceptual level by the non-domain professional far more readily than when expressed in the terms of the software engineer.

2.1.4 Verification using LTSA

Given an FSP model, the LTSA tool can perform deadlock analysis and check for progress violations. Deadlock analysis is performed automatically - given the FSP description LTSA needs no extra information. LTSA can examine the process (or composite process) to ensure that deadlock will not occur by exploring the state-space and testing for a state that has no outgoing transitions (Magee and Kramer 1999, p108).

The absence of such a state indicates that the process is deadlock free. The LTSA tool produces a trace of the actions that lead to the deadlock state (if it does occur), so that the modeller can identify the flaws in the proposed design.

Progress violations can be fired by default behaviours or by user-defined properties. Integrated into a composite process, a property becomes a guard on the behaviour of the process. A property must be defined by the user and explicitly declared in the FSP description. An example property is reproduced from Magee and Kramer (1999, p133):

```
property POLITE = (knock -> enter -> POLITE).
```

This property has an associated graphical representation (of the state space) as presented in *Figure 2 LTSA representation of the POLITE property*. In Figure 2, the '-1' state is a common error state; that is, this is the state that the process will arrive at if the property is violated. Composition of a property with a target process defines the property over the target process.

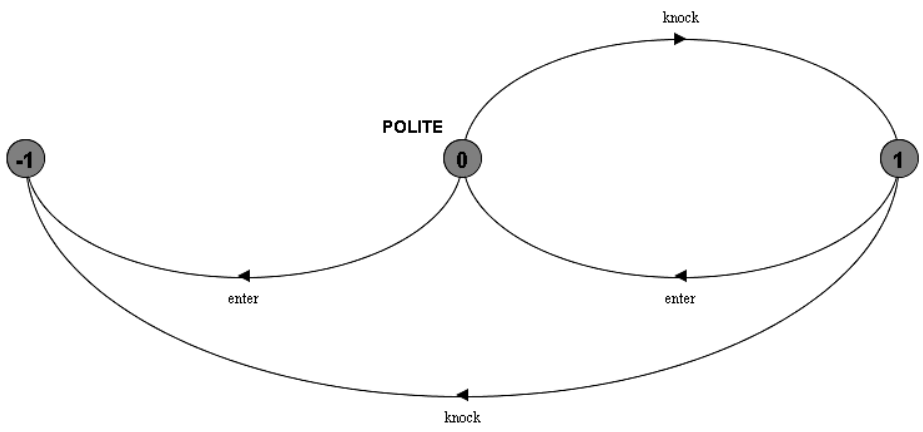

Figure 2 LTSA representation of the POLITE property

Attempted transition to state 1 without executing the 'knock' action leads to an error state – exposing violation of the property POLITE.

In addition to being able to define progress properties, FSP allows the definition of liveness properties. Liveness properties assert that the actions defined in the property will occur either at-once, or at some-later time during the execution of the process over which the property is defined (Magee & Kramer, 1999 p121).

2.1.5 The Darwin Architectural Language

Darwin is an architectural description language that is modelled in the process-calculus of Milner (1980). The process algebra FSP and the animation tool LTSA are also derivative of Darwin (Magee et al 1995, p137).

Darwin is strongly directed at distributed systems. Primitive components are initially modelled as parallel components of more complex hierarchies. A primitive component in Darwin is modelled to a behavioural specification rather than to a structural description (p138).

Magee and Kramer believe that LTSA and FSP have the flexibility required to be used for modelling 'architectural properties' (1999, p239) of systems. They contend that this is so because the abstraction process involved in creating an FSP model is not so concerned with the detailed operation of a process, but with the structure of components and the interactions of components with other processes and components (p239).

2.2 Architectural Description Languages

UML is a pervasive modelling language that may be the language of widest applicability to diverse architectural models (Medvidovic et al, 2002 p4).

Medvidovic et al propose a set of requirements that would form a minimum set of requirements for assessing UML's suitability to software architecture models (p6):

- Structural topology of a system
- Style, i.e. standard vocabularies, generic behaviours, recurring topologies
- Behaviours of systems
- Component interaction paradigms
- Constraints over all aspects of a system

These requirements lead to three strategies for modelling software architectures, firstly, use UML as it is defined, and generally used, by the software engineering community; secondly, constrain UML using native UML extensions; and thirdly, extend UML to explicitly support new paradigms introduced from architectural paradigms (p11).

Medvidovic et al assess the first strategy as satisfactory due to the major advantage that existing knowledge of UML is sufficient to work with architectural descriptions. Existing UML modelling tools would need no modification or extensions. A major disadvantage though is that modelling architectural constructs would require 'implicit' maintenance by the designer (p12).

The second strategy is deemed to have potential as the breadth of UML defines a set that is a superset of architectural concerns. Partitioning architectural constructs through the use of extension mechanisms and Object Constraint Language (Rational et al, 1997) would make explicit the architectural constructs. There is concern however that "it would be difficult to fully and correctly specify the boundaries of the modelling space" (Medvidovic et al, 2002 p12).

Medvidovic et al initially consider the third strategy as 'tempting'. However, the third strategy is quickly disregarded due to the complications of “augmenting UML” (p13). Augmentation of UML would provide a modelling space for every possible ADL to be incorporated as a subset of UML. The negative aspect of this is that it would require learning new rules and constructs for every variant of an ADL, and would also mean extending every UML tool to also interact with numerous new definition languages (p13).

Given the broad range of ADLs that are in existence, and the lack of current support for the broad variety of ADLs under UML, another solution that is discussed by Medvidovic et al is that of architectural interchanges (p47). The idea of an architectural interchange is that systems modelled under one ADL can be automatically transformed to another ADL. This requires each ADL to have a well-defined formal semantics. This also requires each ADL mapping to have mechanisms that enable transformations in the situations where there is not a clear mapping between equivalent or partially-equivalent concepts (p47).

Of the ADLs that are in existence, some are proprietary, and others target a specialised domain such as avionics. The specialised domain ADLs are hard to transfer into other practice due to their esoteric nature. The proprietary ADLs are hard to transfer into practice because they are a closed specification. There are a number of ADLs that have come out of university research which receive treatment and input from the wider, open community (Bass et al, 1998: p269).

Three ADLs that are accessible are Wright, C2, and Rapide. Wright was created “to support more direct specification and analysis ..., specifications (in Wright) are based on the idea that interaction relationships among components of a software system should be directly specifiable as protocols that characterize the nature of the intended interaction” (Shaw & Garlan, 1996 p208). C2 is also aimed at distributed systems. C2 models “software connectors that transmit messages between components, while components maintain state” (Medvidovic, 2002 p14). C2 demands that “notifications sent from a component correspond to its

operations, rather than the needs of any components that receive those notifications. This constraint ... ensure(s) substrate independence" (p15) enabling component reuse and dynamic configuration evolvement. Rapide is an ADL that has a behavioural model based on partially-ordered sets. Rapide components are principally specified in terms of events (p38).

2.3 Software Development Lifecycles

The Rational Unified process (Booch, Rumbaugh & Jacobson, 1998 p449) draws on developments in software engineering - building on methodologies such as the Waterfall Model (Sommerville, 1995 p9) and Boehm's Spiral Model (p13). While the spiral model is touted as a generic process to 'subsume' all previous common methodologies in wide use (p13), the rational unified process is even more so oriented to wide adoption (Booch, Rumbaugh & Jacobson, 1998 p449). Like many methodologies, the rational unified process is an iterative process, whereby models are developed over four phases; the four phases form an iteration of the process. Each phase is further sub-divided into *iterations*, an *iteration* being a "complete development life cycle; from requirements capture in analysis to implementation and testing" (p451).

The four phases in the rational unified process are Inception, Elaboration, Construction, and Transition. Inception is the early stage of a project, where ideas are brought together; Elaboration actively seeks requirements, and develops design models. Construction is the principal phase for Implementation of the solution, and Transition is the deployment phase. The iterations in a phase have differing weights depending upon the phase. Earlier phases have more modelling, while later phases have more testing (p451). Navigation through all four phases is regarded as one complete development cycle leading to a software release (p453).

This thesis is focused on iterations within the Inception and Elaboration phases of a development cycle.

2.4 The Unified Modelling Language and Software Architecture

The Unified Modelling Language (UML) grew out of collaboration between numerous energetic efforts in modelling languages. Booch's *Booch Method* (Booch, 1993), Rumbaugh's *Object Modelling Technique* (Rumbaugh et al, 1991), and Jacobson's *Object-Oriented Software Engineering* (Jacobson, 1992) were variously combined to create UML. Recognising that each language had its own pros and cons, each borrowed from the other, leading to a fusion in practice that has become the Unified Modelling Language (Booch, Rumbaugh & Jacobson, 1998 pp xix).

2.4.1 UML

UML models use diagrams to express relationships between things. There are four kinds of things in UML - Structural, Behavioural, Grouping, and Annotational things (Booch, Rumbaugh & Jacobson, 1998 p18).

Structural things are further decomposed into Classes, Interfaces, Collaborations, Use Cases, Active classes, Components and Nodes. These seven 'things' form the set of structural elements in an UML model (p20).

Behavioural things express the space and time dynamics of an UML model. Behavioural things are mostly interactions or state machines. An interaction is a "set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose" (p21). A state machine is an elaboration of the set of states that may be progressed by any identifying thing. State machine models include states, transitions, events and activities.

Grouping things are used to organize UML models, primarily through the use of packages, which define a set of included components (p22).

The last type, Annotational things, form comments ancillary to the model. Comments help to elucidate and clarify any aspect of the model such as rationale or other detail. Annotational things are mostly expressed with Notes (p22).

UML Relationships have four types: Dependency, Association, Generalisation and Realisation. Dependency relationships indicate the case where a change in one thing will affect the state of another thing. Association relationships indicate a link of some type, often these types are aggregation relationships. Generalisation relationships indicate a subset/superset relationship, where the generalised thing is of the superset and the subset is a specialisation. The realisation relationships imply an expectation from one thing to another, forming a contract.

Most ideas are presented in UML through a diagram, there being at least nine common types of diagrams: Class, Object, Use Case, Sequence, Collaboration, State-chart, Activity, Component and Deployment (p22-24).

2.4.2 Software Architectures

In *Software Architecture: An Emerging Discipline*, Shaw and Garlan posit:

"Good Architectural design has always been a major factor in determining the success of a software system. However, while there are many useful architectural paradigms (such as pipelines, layered systems, client-server organisations, etc), they are typically understood only in an idiomatic way and applied in an ad-hoc fashion."

(1996 p129)

Shaw and Garlan do not see Software Architecture as a mature discipline, reasoning that the lack of formal underpinnings, lack of acceptance of codified models across the industry and idiosyncrasy among comparatively similar definitions (p15). They believe there are four activities driving the emergence of a software architecture discipline:

- The development of architectural description languages. The intent of an architectural description language is to clearly and unambiguously transfer the knowledge of a specific architecture between separate parties. An advantage of defining architectural description languages is that the language then allows analysis and comparison of the architecture.

- Compilation of software engineering practice that “addresses codification of architectural expertise” (p15). Bringing together accepted principles is an aim of this activity, such results as can be found in *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson & Vlissides, 1995). Grady Booch’s Foreword to this book declares “All well-structured object-oriented architectures are full of patterns” (p xiii).
- The relating of, that is - acceptance of, 'frameworks for specific domains', specifically referring to well-established fields such as avionics where work has proceeded widely and publicly enough that there are sound means and methods to achieve results. The reification of practice in this category does not necessarily exclude input from developments that may occur in developmental, experimental, less-structured, or research practice (Shaw & Garlan, 1996 p15).
- Finally Shaw and Garlan address the formal methods in software architecture practice, suggesting the use of formalism for architecture of a specific system, architectural styles, theories of software architecture, and for formal semantics for architectural description languages (p130).

2.5 Formal Descriptions and Verification

Attempting to verify systems that exhibit concurrency raises issues that are not easily resolved. If we consider sequential programs as a subset of concurrent programs, and note that sequential programs are required to terminate, then we are left with an excluded set of problems that are not solved by sequential proofs.

Solutions for the set of excluded problems demands proving of total correctness, that is, proof that the program will halt. Verification of the excluded problems thus requires behaviour analysis to meet this end.

Total correctness problems appear to lack a definitive algorithm to solve them, prompting Dijkstra (1975 p457) to conjecture upon *which part of* 'the programming activity can be regarded as a formal routine, and which part of it seems to require "invention" '?

Choosing behaviour analysis tools is difficult in itself - the wealth of differing notations, formalisms and tools charts a rich vein.

Holzmann has deep questions for automated model-checkers. He asserts that Turing's 'Halting Problem' proves that a model-checker would be incapable of checking itself for logical correctness. This assertion is the basis for Holzmann's question (1998 p103) "...what makes us think we can write a model-checker that can establish the logical correctness of other programs?"

The answer is that model-checkers operate on 'well-defined abstractions of programs', not upon arbitrary programs (p103). This at least is re-assuring. Choosing the right level of abstraction for the task at hand is the key to ensuring productivity with model-checkers, or at least between having a 'tractable model with provable properties, and an intractable model ... only amenable to simulation and manual reasoning" (p104). Holzmann extends this line of thought to the purpose and expectations that initially inspire modelling. He states (p104):

'... the construction of the model is to attempt to disprove the correctness of the chosen solution: to find logical flaws in the reasoning that produced the design .... the model establishes a refutable statement about a design.'

Holzmann validly puts forward that, given advances in model-checker technologies and hardware capabilities, if a problem were to become intractable, then “the design itself is insufficiently understood to be verified, let alone be implemented” (p108).

There are many questions regarding the worth of modelling for verification and validation. Brinksma suggests two schools of thought are converging: Mathematical Reductionism - the view that 'software can be seen as formal objects' - and Experimentalism - a 'phenomenological view ... (where) systems ... are constructed on an experimental basis ... (and) adapted on the basis of ... knowledge and experience' (1992, p33). This equates to verification by mathematical proof - or validation by testing. By composing these mechanisms

Brinksma arrives at a formal framework for validation that incorporates 'the formalization of the concept of observation' (p43).

The extremes of observation in practice account for black-box testing and white-box-testing, i.e. “no versus all structural detail of the implementation under validation are observable” (p43) therefore the notion of *approximate correctness* would encompass relative correctness weightings for different properties of a verification or validation subject (p49).

2.6 Validation-Led Development

Lakos and Malhotra (2002) have argued that it is possible to refine textual descriptions into programs through a sequence of developments led by a process of specification validation. The motivation for performing this action during the development process is primarily directed at ensuring that the eventual outcome of the implementation is satisfactory to the end users needs and expectations.

Thus, validation becomes an early focus of the software effort (Lakos and Malhotra, 2002: p57). They emphasise that object-oriented methodologies help to bridge the gap between specification and implementation. Object-oriented design and programming methods assist the bridging process due to a clean mapping between problem space and solution space.

For static information models the well-known and well-understood processes of object oriented methodologies are effective for this task (p57). The dynamics of system models are not simple to describe as formal specifications, nor as informal descriptions. Where complex lifecycle behaviour is captured accurately at the specification stage, there is often difficulty in translating dynamism into sequential-based programming languages (p57). Lakos and Malhotra reason that it is possible to adopt a methodology that 'retains and implements lifecycle' information as the project 'moves from analysis to design and finally to the programming stage' (p58).

A principal concern of the methodology is that object-oriented design tends to capture the behaviour of entities as atomic actions, rather than state change being dependent upon the state of coupled object interactions.

To verify that any objects' states remain valid in respect to the specification, each object must be tested against a 'plausible sequence of calls' (p59). The greater the complexity of the object being modelled, the more difficult this task becomes - not only to test plausible sequences - but to test all possible sequences. Lakos and Malhotra have developed a methodology which they use to present an implementation of a case study problem.

The purpose of this thesis is to focus on the translation of Lakos & Malhotras' informal *problem description* into the FSP language. Verification of the model as provably correct in respect to concurrency issues, and validation of the model to the satisfaction and expectations of the end users, is the integral task of this thesis.

3 Research Question

The central question of this thesis is to discover if a formal analysis method applied to the design stages of the Software Development Lifecycle can contribute to 'better' models, and therefore better final products. By 'better' is meant correctly functioning software as in 'conditionally provably correct' (verified), and also correctly specified software in regards to the Client's specifications (validated).

This section aims to present the methodology that is explored in this thesis in a structured fashion abstracted away from any particular Case Study or problem instance.

Application of FSP (section 2.1) to the design stage of the Software Engineering Development Lifecycle is detailed in section 5 (Research Process). Section 3.1 explores the idea of 'models' and their use and impact in the Software Engineering Lifecycle. It is hoped that FSP will have significant impact in the elucidation and correction of errors; to set the ground for a discussion of errors, the principal phases of error generation in the context of Validation-Led Development are presented in section 3.2.

To examine the potential role of FSP analysis as applied to a specific problem it is necessary to examine the methodology into which FSP-analysis is to be inserted, component-wise. The motivation for utilising the FSP process as examined in this thesis arises from the Validation-Led Development methodology presented by Lakos & Malhotra (2002). An exposition of Validation-Led Development is presented in section 3.3. Validation and verification of software design using FSP is examined in more detail in section 3.4. This section (3 Research Question) describes the complete methodology, and addresses closely aligned concerns. A summary of this section can be found in section 3.6.

3.1 Software Engineering Model

Software Development Lifecycles are discussed broadly in section 2.3. In this section attention is given to the stages of the SDLC that will be most impacted by the application of FSP analysis.

3.1.1 Client Description

For any software development project there is a starting point, a point at which a germination of an idea is first transferred to paper – a point at which 'hand-waving' becomes obligation and concepts become specifications. The initial text description is in prose form, that is, the form of the words - their phrasing and joining - more often describe a situation, or tell a story, than provide a specification. The clients' description may also be directed more towards a perceived solution than to a well-defined problem. The client may be very clear about the end product, a position variously experienced as hindrance or boon – or both – by the developer.

Language, domain specific lexicon, and cultural / societal variations influence the world-view of the client and the developer. The world-view in turn shapes understanding and communication. Though not necessarily causing aberrations in problem descriptions, informality of language may lead to mis-specification of descriptions and solutions.

3.1.2 Developer Model

The input available to the developer is the *client description* (Section 2.3.1). From the *client description* the developer must create the *developer model*. The mapping from *client description* to *developer model* is not a trivial task. Misunderstandings of lexicon, (mis-)use of language, vague and ambiguous meanings all contribute to 'muddying of the waters'. Where the development stages are performed by teams there is also the potential for 'Chinese Whispers' to distort meanings and specifications.

Consequently, the *developer model* can itself introduce errors. Providing mechanisms in the SDLC that validate developer understandings against client wishes may alleviate tendencies to divergence.

3.2 Error Generation and Removal

The propensity for errors to propagate and amplify suggests that a development methodology should strive to minimise errors at each stage of its evolution.

Ideally each stage should output fewer errors than were present before the stage began. Both client and developer have the ability to introduce errors. Errors may arise from incorrect mapping of the developer model, or from issues inherent in the problem description itself.

These latter error types are not exclusively the result of incorrect client descriptions, but may arise while 'filling-in-the-gaps' of otherwise sound client descriptions. The need for developer model and subsequently the Implementation Specification to require greater detail of specification requires that the developer apply knowledge, experience and creativity to selecting the correct set of data structures and algorithms.

3.3 Validation-led Development

In Validation-led Development, Lakos & Malhotra focus on Object-Oriented models to provide clean mappings between output models. They recognise that behavioural analysis is complex, and that translation of dynamic models into sequential programming languages can be problematic (2002, p72). We can build on their methodology of retaining lifecycles through development stages (2002 p58) by formally checking developer models for concurrency issues.

Results from formal verification of concurrency properties aid in validating client descriptions by allowing clearer definitions of the problem solution (section 3.4.2, 3.4.3) to be matched against the problem description (section 3.4.1). The refining of textual descriptions is another source for errors that can corrupt mappings between specifications.

3.4 Validation of Software Design using Finite State Process

This thesis consolidates the verification of behavioural analysis offered by validation-led development and also seeks to extend the Lakos & Malhotra *validation-through-refinement* method. The extension is gained by evaluating the effectiveness of FSP as a component of validation-led methodology.

The application of FSP to the traditional SDLC and to the validation-led development methodology defines certain expectations, benefits and disadvantages. These issues are likely to influence each phase and each deliverable to a different degree. The following is a list of areas of software development lifecycles that will be FSP-influenced. The listed topics include a description of the expectations that arise from the application of FSP.

3.4.1 Client Description

The client description, presented in Section 4.2, is derived from the problem description as given in Lakos & Malhotra (2002, p58), which is reprinted in Section 11.1. A discussion of the derivation is presented in section 6.3.

The client description is a focal point of the evaluation phase of the Research Process, as discussed in section 5.5. During the evaluation phase, output from the iterative application of FSP is to be input to consideration of the initial text description in light of the iterative FSP process.

Refining of text descriptions is described here as a single activity, rather than as a continual process. Justification for this, and also justification for the Transition Tables (Section 11.2) to be regarded as the Developer Model, is offered by way of recognising that the research process in this instance is a closed operation, with no third-party client to whom to address concerns. Models and documentary materials are drawn from Lakos & Malhotra, 2002.

3.4.2 Developer Model

In contrast to the *client description* as somewhat immutable in regards to client interaction, the *developer model* is readily available for direct modification if this is found necessary.

The *developer models'* original form is reproduced directly in Section 11.4. The *Transition Tables* are a set of 5-tuples that specify the *Current State*, *Event*, *Guard*, *Actions*, and the *Next State*. Each 5-tuple also has an identifier.

In the context of this thesis and in the context of Lakos & Malhotra (2002) these terms have the meanings defined in Sections 3.4.2.1 – 3.4.2.2.

3.4.2.1 Transition ID

An identifier for each 5-tuple in the form of a letter followed by a number, eg T12 which identifies Transition 12 of the Lift Lifecycle.

3.4.2.2 Current State

Defines a state from where progression to another state is enabled by a set of guards holding if a specified event occurs.

3.4.2.3 Event

A notification that tests the guard expression defined for a transition and, if the guard holds, allows a sequence of Actions - terminated by entry to a new state.

3.4.2.4 Guard

A guard is the predicate expression that must be met to enable the firing of the transition.

3.4.2.5 Actions

Actions are activities of objects that proceed after the matching of the guard in a transition, and before entry to the new state.

3.4.2.6 Next State

The object will complete the firing of a transition by assuming a new state. The new state will be subject to a different set of transitions than the state that has just been left.

The *developer model* is the departure point for the FSP model; full implementation of the developer model in Finite State Process will hopefully lead to the exposure and resolution of errors in the developer model and also guide the formation of the corrected model.

3.4.3 Implementation Specification

The final output from FSP-analysis is coupled with a final evaluation phase to form the Implementation Specification – a set of documents communicating the formally-verified behavioural aspects of the object lifecycles and their interactions, and also an object model based upon the formally verified model.

Mapping back to an object-oriented domain from the FSP process calculus is included in this exploration of the efficacy of FSP for two reasons: firstly, Unified Modelling Language (Section 2.4) is widely accepted and so provides a common entry point to the FSP model, and secondly, the inclusion of an FSP analysis component to an existing SDLC demands that there is proven utility with a minimum of digression.

3.4.4 Error Removal through Iteration

Execution of the iterative process of relabelling in FSP is expected to have two principal phases. The first phase consists of translating the components and their

interactions from the developer model into the FSP description language. The translation effort enters the second phase of informing the developer model by the evolution of the FSP program.

The process of iteration potentially induces a gray-zone between these two phases, a haphazard space where adjusting the one model affects the other. To attempt to remain faithful to the developer model in modelling at this juncture is to maintain the input specifications as immutable, as mirrored by the FSP model, *until such time* as the *correctly-mapped* output specifications are found to be in error.

Adopting this stance it is hoped that errors in the mapping of the developer model to the FSP language will be removed early, leaving the latter duration of the iterative process to examining issues in the developer model and in the client description. Issues that can reasonably be expected include mis-specified descriptions, mis-translated developer understandings, and inherent concurrency issues arising from the interaction of discrete object types.

3.4.5 Description – Model – Analysis Cycle

The crux of the FSP analysis phase is now fully outlined in the preceding four sub-sections. To recap, from the client's initial text description the developer derives a model of the problem solution. The developer model is then treated to formal analysis through being translated to the FSP process-calculus. Interaction between the developer model and the FSP model forms the basis for iterative application of the FSP checking tool, and consequently, re-specification of the FSP-model, the developer model and the initial client description.

3.4.6 Corrected Model

The term 'corrected model' has been used loosely so far in the explanation of the FSP methodology. While conveniently used to refer to an arbitrary set of documents presumed to have an 'aura of infallibility', it is rather the case that what is being referred to exists only as an abstraction – as this term specifically indicates the most recently iterated over FSP model.

Examination of exit conditions from the iterative FSP-analysis is necessarily performed transparently to enable justifications of verification to be made of the corrected model.

3.5 Pre-FSP Object Model

As discussed in sections 3.4.1 and 3.4.2, the initial documentation of client description and developer model is input to this research process. The creation of a first-stop, rudimentary object model also has certain benefits. An object model created prior to the FSP-analysis stage will not be 'coloured' by the FSP-analysis, and so will serve as a reference point for later comparisons of the FSP-produced object model.

The object model created before the iterative FSP-analysis is required to serve as a comparison to the Implementation Specification object model produced as a result of the iterative FSP-analysis, for the purpose of evaluating the FSP generated object model in light of an object model created with less stricture.

To begin the initial FSP translation it is also necessary to perform some decomposition of the client description, the pre-FSP Object Model adequately provides this initial decomposition.

3.6 Summary of Research Question

Summarisation of this chapter discloses a number of activities that are to be performed in the production of this thesis. Firstly, apply *FSP to verification of behavioural analysis*, and secondly, apply *iterative–FSP to the validation-led methodology*.

This can also be expressed as an ordered list of tasks that repeats over steps 3 – 7 until exit conditions are satisfied:

1 Create an UML object-structure model from the client description,

2 Implement a developer model in FSP,

3 Formally analyse the FSP model using the LTSA tool,

4 Document, evaluate and rectify errors,

5 Refine the developer model,

6 Build-up the corrected model,

7 Refine text descriptions,

8 Create an Implementation Specification that consists of a set of behavioural diagrams and an object model.

The application of this methodology to a specific task is specified in the next section: *Section 4, Problem Statement*.

4 Problem Statement

The central task of this thesis is to apply a specific methodology component to a given problem and then evaluate the results.

Given that the methodology component seeks to redress errors in models of the problem space and also in models of the solution space, particular emphasis is to be placed on analysing the types of errors found.

The results of evaluating exposed errors are to be used to redefine, or if necessary, to correct initial descriptions of the Case Study problem, and also to correct (or refine) models of the proposed solution.

4.1 Apply FSP to a Case Study Problem

While applying FSP-analysis to a specific problem, documentation will be collected including FSP code-listings of processes and composite processes, details of errors as revealed by the LTSA tool, descriptions and comments upon these errors, state space data and object models.

The initial text description of the Lift Problem Case Study (Section 4.2) is derived from the description as specified in Lakos & Malhotra (2002). Section 6.3 discusses the derivation of the description.

4.2 Case Study: The Lift Problem

A building is serviced by several identical lifts.

These lifts travel between the floors of the building. Each floor, with the exception of the ground and the top floor, has two call buttons - one for each direction of the travel. For obvious reasons, the ground and the top floor have only one button.

Passengers arriving at a floor press the call button appropriate to their direction of travel. They wait in a first-in-first-out (FIFO) queue for their turn to enter a lift.

When a lift arrives and the doors open, the current passengers destined for this floor get out, and then as many waiting passengers as possible get in. These embarking passengers press appropriate buttons for their destinations. When the lift arrives at their destination, they get out.

An idle lift continues to be inactive and stationary at a floor, with the door closed, till it is activated in response to a call from a newly arrived passenger. An idle lift, when activated, begins to travel towards the calling floor and serves other passengers.

An active lift continues to move upwards and downwards serving the waiting passengers until it finds no further remaining work. At this stage it becomes idle. A lift going upwards halts at various floors to drop passengers and to pick up new passengers. It does not change its direction of travel until it reaches a floor where it is no longer carrying a passenger, and where there are no waiting passengers on any floor above.

The lift may change direction in order to service the passengers waiting to travel in the other direction. If this also yields no further work for the lift, the lift enters its idle state. An analogous behaviour is shown by a lift travelling downwards. Each lift has as many destination buttons as there are floors in the building.

5 Research Process

Section 5.1 describes the development of a model of the Lift System, following the methodology outlined in Section 3.4. A description of the phases of the development is provided along with examples of the model in various forms. Evaluation of errors and analysis of the documentation is treated in section 5.2. Evaluation is not performed in isolation from the iterative process as the grouping of the documentary discussions and materials may suggest. Translation of the corrected model to an Implementation Specification is described in section 5.3. Open communication with the client is a desired factor that would allow this phase to be interpolated throughout the cyclic process, each issue that is pertinent to the client is considered in section 5.4.

Section 5.5 discusses the effect on the developer model of FSP iteration, and the FSP-influenced analysis process as a whole is recounted in section 5.6.

This section refers to the software tool LTSA and the language FSP as presented in Section 2.1 and as such presumes familiarity with the language and the tool.

5.1 Development of Lift System using FSP

Decomposition of the lift problem enables early entry into the problem space. Sections 5.1.1 - 5.1.3 are accounts of the modelling phase before the commencement of iterative development. Identification of type candidates and decompositions of actions from the Lift Problem is described in Section 5.1.1. Decomposition provides the building blocks for construction of the pre-FSP object model (Section 5.1.2).

The pre-FSP object model is firstly treated to abstract modelling and then detailed state modelling, before being treated to iterative FSP application which also provides the broader abstractions used at the initial FSP modelling stage (described in Section 5.1.3). Section 5.1.4 discusses the composition of the components, their shared interfaces, and FSP-specific mechanisms such as relabelling.

Testing at the start of each cycle leads to error identification (section 5.1.5). The resolution of the errors through remodelling has two parts:

1 Individual component remodelling which is described in section 5.1.6

2 Composed component remodelling, also described in section 5.1.6.

These two parts may have to both be performed as required to resolve the error at hand, alternatively, only one of the two may be necessary to be performed to resolve the error at hand. The completion of resolution of the error also completes a cycle of the iterative FSP analysis and so returns us to examining a newly revealed error. Each cycle generates documentation which is examined in Section 5.1.8. Descriptions of the exit conditions from this cyclic development of the FSP model is given in Section 5.1.9.

5.1.1 Identification of Components

The initial text description (section 4.2) supplies details of the actors in the Lift System. Besides the actors we may also extract actions that are performed by the actors. Actors themselves may be split into two broad types - passive and active.

Active actors are able to initiate events; they are also able to respond to actions with deliberated action, for their own purposes. Passenger objects are identifiable as Active actors, but they can exhibit passive and reactive behaviour as well as active behaviour.

Passive actors exhibit state change through the actions of other actors, for example, a floor has state change based on the comings and goings of Passengers.

Entities may inhabit any part of the range between the two extremes of active and passive. An entity will mostly display variations of passive / active behaviour throughout its lifecycle. For example, the Lift entity is variously passive or active, while its components - the buttons and doors - are generally passive.

A Building entity primarily exists as a passive container object. Buildings aggregate Floors and Lift Systems as first-class objects, whereas Buttons and Doors may be aggregated via active entities.

Actions are mappable to a multiplicity of relationship categories; the relative exposition of object properties desired in the final model determines the patterns of actions selected to exhibit object behaviour.

5.1.2 Development of pre-FSP Object Model

The distinction between passive and active actors contributes to placing components into an initial object model. State variables can be elicited from action descriptions along with suggestions to their parentage, association or aggregation. There are many choices for binding objects; a potential solution is depicted in *Figure 3 Pre-FSP Object Model of the Lift Problem*. Figure 3 portrays an object model of *one possible representation only* of the *structure* of the entire Lift System.

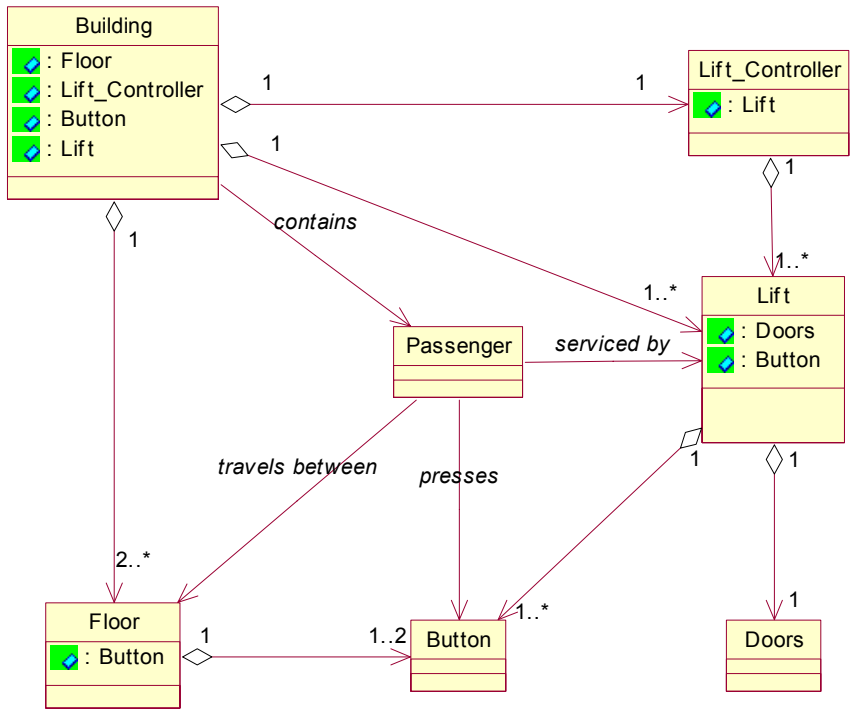

Figure 3 Pre-FSP Object Model of the Lift Problem

An initial modelling of the client description provides insight into the structural and behavioural properties of the problem space – principally exposition of the containment relationships between the actors (active and passive).

This model is discussed further in Section 7.2 where discussion of the pre-FSP object model is made in reference to the FSP object model. This model (Figure 3) is not, at this stage of the methodology lifecycle, intended to be a finished model, rather it is a representation of an early foray into object model creation representative of *early stage* iterative design development.

5.1.3 Modelling components in FSP

Derivation of an FSP-model commenced with the identification of the principal processes. The first candidates were the Lift objects and the Passenger objects.

At this stage it is convenient to regard the Lift Controller object functions as subsumed to a function of the Lift object, which also has the effect of limiting state-space explosion (Section 7.3). This is useful for modelling a single Lift. The Lift Controller / Lift relationship and the abstraction of the Lift Controller are discussed in section 6.3.1.

The FSP processes for these actors evolved from two distinct approaches to their formation. The first approach was based on the analysis as described in section 5.1.1 and 5.1.2.

Removing state from the FSP process descriptions allows distillation of the actors to their actions which exposes their lifecycles clearly, albeit broadly. Lifecycle interaction is readily apparent in this stateless approach which enables the rough construction of general models onto which finer detail can be imposed.

The listing in Section 11.2 is the initial FSP description for a Lift, while the listing in Section 11.3 is the corresponding description of a Passenger. Both of these listings are modelled on the developer model (Section 3.4.2). They are FSP implementations of the developer model, with state variables and guards only marginally introduced.

In these listings the early stage of the FSP translation can be appreciated from the abundance of guards that are blocked out (to mirror the developer model) – they only hold literal values. These ‘dummies’ are subsequently expanded to sets of boolean expressions to constitute the guards as defined in the developer model,

for example, the following FSP code from the final FSP corrected model (Section 11.6) is mapped from Transition 8 of the developer model (Section 11.4).

```
//      T8              no one wants to get out at this floor
//                      (j==0)
//                      AND lift is not full
//                      (p<MAX_PASSENGERS)
//                      AND there is a waiting passenger on floor
//                      for
//                      current direction
//                      (switch to test)
if (j==0 && p<MAX_PASSENGERS && d==UP && f<NUM_FLOORS) then
(
        [BTN_UP][f].seek_button[i:0..NUM_PASSENGERS] ->
        if (i>0) then
        (
                [BTN_UP][f].off ->
                enter_lift[f] ->
                PICKING_PASSENGERS[p+1][f]
        )
        else
        //      other transitions may be valid
                DROPPING_PASSENGERS_T9[p][f]
)
```

There are several advantages to adopting the developer model. Firstly, the developer model is specified with respect to 5-tuples of start-state, event, guard, action(s) and next-state, which are mappable to FSP with some alteration. Secondly, drawing on the validation-led specification places the application of FSP and formal testing cleanly within the framework of the validation-led methodology.

The code structure as it is defined in FSP (mapped from the developer model) is to be maintained through the iterative modelling process until such time as the generation of errors points to issues *within* the developer model. At this time modification of the developer model becomes a requirement for attaining correctness.

5.1.4 Composition of Individual Elements

The composition of the Lift and Passenger processes requires more actors than just Lifts and Passengers to describe the full problem. For example Floors are required in the model *in some form* so that we can be specific about Passenger and Lift movements.

Buttons (modelled as queues) equip the FSP model with state, event-handlers and also as a mechanism for dispatch so that the Lift is able to determine its *next action* after each state-contextualised event. Interaction between Lifts and Passengers is also achieved through direct synchronisation of actions using the FSP relabelling construct (Section 5.1.7).

5.1.5 Test Model using LTSA

Iterative testing of the FSP-model is dictated by the nature of the LTSA checking-tool (Section 2.1.2). LTSA traverses the state-space defined by the model and reports the first safety error or the first progress error - or both - that it encounters but no more than one of each type. To continue development the issue that is the cause of the violation has to be addressed and resolved. Variations on a theme – that is, fixing the symptom - can be time-consuming when the exact and correct root cause of an issue is misidentified.

When a safety violation and a progress violation present at the same time (and, they do not reflect the same root cause) the decision as to which error to address has largely been a matter for prudence. Resolving the progress issue rather than the safety issue was often seen to expose a different progress issue, and had the effect of a short-term 'burying' of the safety issue, which would then emerge at a later time.

An example error trace is provided below. The specific model that generated this trace is LiftAndPassenger_D6.lts. This model is discussed further in Section 5.2.1.

```
Composition:
LP = p.1:PASSENGER || LIFT(btnUp,btnDown,dptCount) || btnUp.1:BUTTON ||
btnUp.2:BUTTON || btnDown.2:BUTTON || btnDown.3:BUTTON || dptCount.1:BUTTON ||
dptCount.2:BUTTON || dptCount.3:BUTTON
State Space:
 22 * 882 * 3 * 3 * 3 * 3 * 3 * 3 * 3 = 2 ** 29
Analysing...
Depth 25 -- States: 217 Transitions: 445 Memory used: 4482K
Trace to DEADLOCK:
        p.1.arrival.1
        p.1.passenger.1
        p.1.call.1.1
        delay.1
        door_is_open_i.1.1
        dptCount.1.seek_button.0
        btnUp.1.seek_button.1
        btnUp.1.off
        p.1.enter_lift
        p.1.entered_lift.1
        p.1.press_dest.1
        pause
```

```
        door_is_closed.1.1
        dptCount.1.seek_button.1
        door_is_open_i.1.1
        dptCount.1.seek_button.1
        p.1.destination_reached.1
        p.1.left_lift.1
        p.1.arrival.1
        p.1.passenger.1
        dptCount.1.seek_button.0
        btnUp.1.seek_button.0
        btnDown.1.seek_button.1
        btnDown.1.off
Analysed in: 180ms
```

5.1.6 Remodelling of Components

Remodelling of the components of the FSP model is necessary to resolve safety and progress issues that have been revealed by LTSA. The following code listing is excerpted from a documented error case that arose during the iterative FSP-modelling process.

The model that generated this error is identified as *LiftAndPassengerPV2.lts*.

```
LOOK_DOWN_INTERNAL[p:OCCUPANTS][f:FLOOR][i:FLOOR][d:DIRECTION] =
(
        pauser_i_d ->
        if ( (i-1) > 1 ) then
        (
                [DP_COUNT][i-1].seek_button[j:0..2] ->
                if (j>0 && f-1 >= 1) then
                        (adelay_i_d[f][i] -> IN_TRANSIT[p][f-1][DOWN])
                else
                if (i-1 > 1) then
                        LOOK_DOWN_INTERNAL[p][f][i-1][d]
                else
                        LOOK_DOWN_EXTERNAL[p][f][f][d]
        )
        else
                LOOK_DOWN_EXTERNAL[p][f][f][d]
),
```

The error here is that the guard for the first conditional test is 'off-by-one', i.e. a range-check error, and so it blocks the recursion of the internal 'button_seek' from reaching the boundary case. This error produces the following progress violation trace when run through LTSA (Note that optimisation of expressions has not been performed at this stage of development, i.e. although `f-1>= 1` can be simplified to `f>1`, the first expression maintains the ‘logical model’ of the processes state variables. Simple optimisation was performed at the transition between FSP models PassengerAndLift.lts and LIFT_SYSTEM.lts).

```
Progress Check...
```

```
-- States: 30 Transitions: 56 Memory used: 4699K
Finding trace...
Progress violation for actions:
{adelay_e_d[2..3][3], adelay_e_u[1][1], adelay_i_d[2..3][3], adelay_i_u[1..2][1],
at_next_floor[1..3][0..1], {btnDown, btnUp}[1..3].{off, seek_button[0..2]}, delay,
delay[1..3], delay2, door_is_closed[0..1], {door_is_open,
door_is_open_i}[1..3][0..1], dptCount[1..3].seek_button[0..2], idling[1..2],
it_adelay_e_d[2..3][2..3], it_adelay_e_u.{[1][1..2], [2][2]}, it_pauser_e_d,
it_pauser_e_u[1..3], p[1].{arrival[1..3], call[1..3][0..1],
destination_reached[1..3], enter_lift, {entered_lift, left_lift, passenger,
press_dest}[1..3], wait, waiting_in_lift[1..3]}, {pause, pauser_e_d, pauser_e_u,
pauser_i_d, pauser_i_u}, press_call[1..3], rit_adelay_e_d[2..3][3],
rit_adelay_e_u[1][1..2], {rit_pauser_e_d, rit_pauser_e_u}}
Trace to terminal set of states:
       p.1.arrival.3
       p.1.passenger.3
       p.1.call.3.0
       delay.1
       at_next_floor.2.1
       dptCount.2.seek_button.0
       btnUp.2.seek_button.0
       btnDown.2.seek_button.0
       it_pauser_e_u.2
       btnDown.3.seek_button.1
       it_adelay_e_u.2.2
       at_next_floor.3.1
       dptCount.3.seek_button.0
       btnUp.3.seek_button.1
       btnDown.3.seek_button.1
       btnUp.3.off
       p.1.enter_lift
       p.1.entered_lift.3
       press_call.1
       pause
       door_is_closed.1
       pauser_i_u
       pauser_e_u
       pauser_i_d
       dptCount.2.seek_button.0
       pauser_i_d
       pauser_e_d
       btnDown.2.seek_button.0
       pauser_e_d
Actions in terminal set:
       idling[3]
Progress Check in: 1093ms
```

Tracing the sequence of events shows recursion of the seek-button blocking before reaching the first floor. Correcting the range-check and then testing again reveals two new errors, one each of Safety and Progress type. Examination of these new errors reveals that they are not connected to the previous problem, indicating that we have resolved the error in the FSP model *LiftAndPassengerPV2.lts*.

Precedence is generally given to resolving safety issues before progress issues. The safety issue that is now raised is a deadlock (*LiftAndPassengerD3.lts*) as a result of an incorrect FSP-implementation of the interaction between the Lift and the External Buttons. This specific issue is discussed in Section 5.2.1. Resolution of concurrency errors will resolve issues that are exposed through the interaction

of processes; it may be that only one of the involved processes requires alteration to rectify the issue.

5.1.7 Composition, Relabelling and Hiding

Early versions of the Button structures were modelled on the PORT model (Magee & Kramer, 1999 p215). The FSP model for a PORT of two messages has this form:

```
set S = {[FLOOR],[FLOOR][FLOOR]}

PORT = (send[f:FLOOR] -> PORT[f]),
PORT[f:FLOOR] = (send[f2:FLOOR] -> PORT[f2][f]
                      |receive[f] -> PORT),
PORT[f2:FLOOR][f:FLOOR]       = (receive[f] -> PORT[f2]).
```

This Button structure subsequently changed to the structure shown in the following listing, which is excerpted from the final FSP model (Section 11.6).

```
BUTTON = BUTTON[0],
BUTTON[i:0..NUM_PASSENGERS] =
(
               when (i<NUM_PASSENGERS) on -> BUTTON[i+1]
       |       when (i>0) off -> BUTTON[i-1]
       |       seek_button[i] -> BUTTON[i]
).
```

The relabelling operator is used in the composition block of the FSP model to fuse an action from a Passenger with an action of a Button:

```
…
passenger[p:1..NUM_PASSENGERS].call_at_ground_floor/btnUp[1].on,
…
```

Finally, the Hiding operator is used in the final FSP model to reduce the number of elements in the Lifts' exposed API:

```
...
/* LIFT Code */
...
\{                              // Hiding operator
choice_T4,
choice_T5,
choice_T9,
choice_T12,
choice_T14,
...
...
```

```
it_l_u_e,
it_l_u_e_opp,
it_l_d_e,
it_l_d_e_opp
}.
```

5.1.8 Documentation and Iteration

A series of individual issues (errors) are documented. The document *Iterative FSP-analysis: Error Documentation Summary* is located in Section 11.5.

Figure 4 summarises the documentation of error *LiftAndPassengerPV16.lts* which was collected during the iterative process of re-modelling components and their interactions:

ProgressViolation16.txt	Yes	Yes	LiftAndPassengerPV17.lts	457	Passenger arrives at floor 2, indicates to travel up. The lift travels up and the door opens. The Passenger enters the lift, indicates to travel to the current floor. The door closes and then the door opens. The Passenger leaves, and then a Passenger arri	Need to add branching to closing door to allow for safe check of external buttons at any floor, before going for walk.

Figure 4 Iterative FSP-analysis: Error Documentation, LiftAndPassengerPV16.lts

The columns represent - respectively - error model file identifier, specification error, development error, solution model file identifier, line number (error model) of root cause of error, description of error, and solution description (or comment). Note that errors may be categorised as an error type from more than one stage of the iterative process – this reflects shared complicity in the error production and/or consolidation.

LiftAndPassengerPV16.lts error documentation consists of the FSP code listing, output error trace (*LiftAndPassengerPV16.txt*), a description of the problem identified and a description of the solution applied. The solution model is referenced by the next error number, i.e. *LiftAndPassengerPV17.lts*, as the 'corrected code' becomes input for the next iteration of testing.

5.1.9 Corrected Model Exit Conditions

Simply put the exit condition for leaving the iterative process is the non-generation of errors from LTSA. The bounds of state-space modelling (Section 7.3) allows only assertions to be made for the quantities tested.

However, the problem space may offer its own natural limits that answer the question. For example, a description for a lift implementation may specify 3 floors and 10 passengers only.

Achieving total state space exploration for the desired scale is computationally expensive. What is more, we can only assert properties when traces complete, i.e. the program halts.

The execution of the iterative process as described in this section ceased when the FSP model was no longer generating errors for the bounded values of passenger and lift that were explored with the LTSA tool.

5.2 Evaluation of Error Documentation

Each error that has been documented is classified by identifying the design phase that *generated* the *root cause* of the error. That is, for each error found by LTSA, attribute that error to the design phase where that error was injected into the model. Identified categories are injection at *analysis*, injection at *developer model* construction, and injection at *client description* formulation. The source for this précis of the error data is the document *Iterative FSP-analysis: Error Documentation Summary* (Section 11.5).

In regards to the temporal sequencing of this error exposition, the presentation of these errors is categorised by the type of error – however the numeric identifier places the error in a linear sequence which allows recognition of *when* in the iterative process the error was exposed.

5.2.1 Analysis errors

The majority of documented errors are generated in the model-analysis stage. The trivial case for this category includes typographical and range-check errors. The less trivial case includes misinterpretation of the developer model.

- LiftAndPassenger Progress Violation

A typographical error, the model compiled but displayed a runtime error.

- LiftAndPassenger Progress Violation 2

  This error resulted from the construction of the *walk algorithm's* (Section 5.3.1) functions as FSP processes.

- LiftAndPassenger Deadlock 3

  Up buttons on the top floor occur in this model as the guards are not placed to logically exclude a transition to a state.

- LiftAndPassenger Deadlock 6

  By tightening the branch points and producing separate paths through situations where illogical (semantic) combinations can be logically allowed in FSP, a similar case to *LiftAndPassenger Deadlock 3* can be avoided, and is the solution to this error.

- LiftAndPassenger Progress Violation 7

  The exit points of the *walk algorithm* that do not fire travel events were at this stage looping back into the *walk algorithm* and so executing a busy-wait loop, waiting for a message from any button to be received. This error found the introduction of a data structure to monitor passage through the *walk algorithm*, and then allow transition out to the IDLING state as per the developer model. Resolving this error tends to re-categorise this issue as a developer model issue rather than as an analysis issue as we are contributing to the developer model's completeness.

- LiftAndPassenger Deadlock 8

  Another illogical button action raises this issue; again a three way branching allows exclusion of the errant transition.

- LiftAndPassenger Progress Violation 9

  The error documentation refers to this error as being a product of 'development', which is distinct from the developer model. The issue here can be regarded as typographical, but actually is a result of placing the wrong code in the right place by not understanding, or – rather, not mapping the developer model clearly or correctly.

- LiftAndPassenger Progress Violation 10

  Tightening a range-checking guard allows the full recursion of the function in this instance. Again, there is the use of 'development' in the error documentation to describe the FSP-modelling process.

- LiftAndPassenger Progress Violation 11

  A loose guard allows this error to occur, creating an illogical floor / button action as in some previous errors.

- LiftAndPassenger Progress Violation 13

  An issue that arose because of the remodelling in *LiftAndPassenger Progress Violation 7*, one of the two new local processes is incorrectly specified.

- LiftAndPassenger Progress Violation 16

  Again, refinement of newly introduced processes leads to safe checking of external buttons, i.e. no attempts to fire illogical combinations of button / floor.

5.2.2 Model Errors

Errors due to inconsistencies in the developer model are categorised as model errors. Errors in the model manifest as analysis errors, however the issue at hand is resolved in the model.

- LiftAndPassenger Progress Violation 3

  The resolution of this error was deferred at this point in favour of resolving the *LiftAndPassenger Deadlock 3* condition. This issue becomes buried but arises again in *LiftAndPassenger Progress Violation 12*.

- LiftAndPassenger Progress Violation 4

  This error is as a result of premature exit from the *walk algorithm*. The solution to this error sufficiently contributes to the developer model by defining the loop-back to the beginning of the *walk algorithm* with a flip in the Lift direction.

- LiftAndPassenger Progress Violation 5

  As a result of not making a check at the current floor after closing the door has the effect of leaving a passenger in a halted lift. Placing a guarded action to check for this condition is a simple solution and is also easy to map back to the developer model as a contributory specification regarding the *walk algorithm*.

- LiftAndPassenger Progress Violation 12

  The *walk algorithm* halts normally, i.e. performs a correct walk, but leaves the passenger waiting for an enter_lift event. This occurs when a passenger is intending to travel in an opposite direction to the lift, and the passenger is at the same floor as the lift. Also there are no passengers waiting at floors in the direction of the lifts travel. Definable as a boundary condition this error is resolved by adding a process at the start of the *walk algorithm* to identify this particular condition.

- LiftAndPassenger Progress Violation 14

  Rather than exposing an error in the developer model or in the FSP model, the issue at this point is that the call received_action is not mapped to occur during the major states of the Lift Process, only during the IDLING process. This is as a result of compartmentalising concerns to allow incremental development of the developer model.

- LiftAndPassenger Progress Violation 15

  Addition of a function in the *walk algorithm* allows the correct handling of a boundary case. The FSP code to handle this state defines another contributory specification.

- LiftAndPassenger Progress Violation 17

  This error is the first that does not exhibit a fatal behaviour in the running of the model. Instead, this error concerns the ‘pointless’ closing of a door when a passenger is still to exit the lift. In defining an optimisation the solution becomes a useful contributory specification as opposed to an essential specification.

- LIFT_SYSTEM Deadlock 2

  This error is fatal however, and reveals that the ordering of transitions T17 and T21 is required to be reversed to disallow a situation where the lift can go off to another floor and enter an unstable state. The swapping of the order of these two transitions is of major contribution to the developer model.

- LIFT_SYSTEM Progress Violation 3

  Variation in the ordering of guards resolves this issue; therefore the contribution is to the developer model in correctly specifying the *walk algorithm*.

- LIFT_SYSTEM_SAFE

  The inclusion of this issue arises from the LTSA tool flagging any unreachable states as a Progress Violation on those states. The unreachable states themselves are the processes explicitly handling the transitions T20 and T21. T21 still survives, but only as a transition to the IDLING state. Removal of the paths to these remnant process-blocks, and the removal of the remnants themselves, resolves this issue.

5.2.3 Description Errors

The last category collects errors that are the result of an issue in the initial text description. There are no direct revelations of errors in developer model that were derived from the initial text description, nor are there any directly revealed errors from the initial text description itself.. This is partly testament to the formality of the process that produced the developer model from the text descriptions (Validation-Led development, Section 2.6). There are contributions that may be made to the initial text description from the errors in the previous two categories however that are of value. These contributions are discussed in Section 7 (Research Discussion), and also in Section 5.5.

5.3 Corrected Model to Implementation Specification

The FSP model that is output from the iterative process is deemed the 'corrected model', as referred to and defined in section 3.4.6. Conversion of the corrected model to an Implementation Specification is a translation process between model representations (as discussed in section 3.1); therefore, this translation is also subject to the phenomenon of injection of non-trivial errors. Translation to an Implementation Specification requires maintaining faithfulness to the FSP model, so that the mapping from the developer model through to the analysis model and mapping of the corrected model to the Implementation Specification model is clearly evident. Representation of the model in the less-formal (compared to FSP) UML domain removes the ability for the model to be found provably correct.

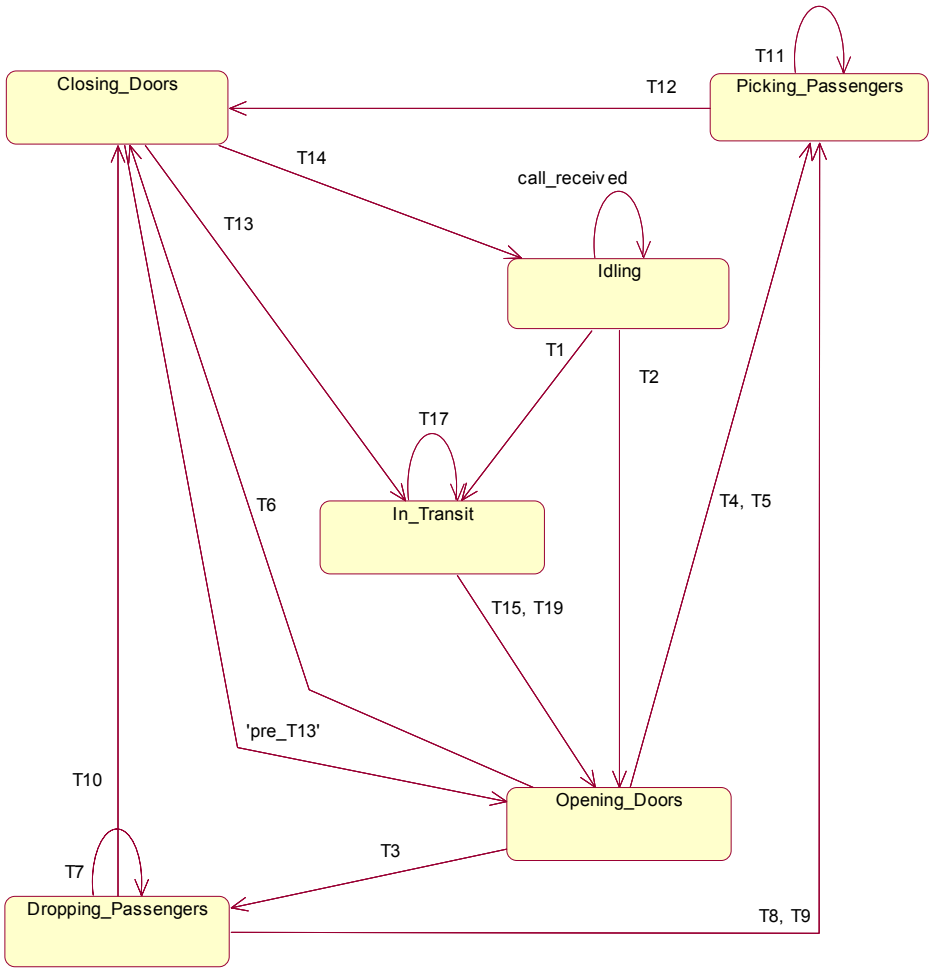

Figure 5 LIFT_SYSTEM_SAFE: Lift object behavioural diagram

Represented in this diagram are the principal states that a lift will occupy at different points of its lifecycle.

A behavioural diagram of the Lift Process is given in Figure 5. Each state in this diagram represents a set of processes from the Lift Object in the FSP model. Diagrams of the behaviour of each set of processes are presented in Figures 7 through 12. An object model of the FSP model is given in Figure 6.

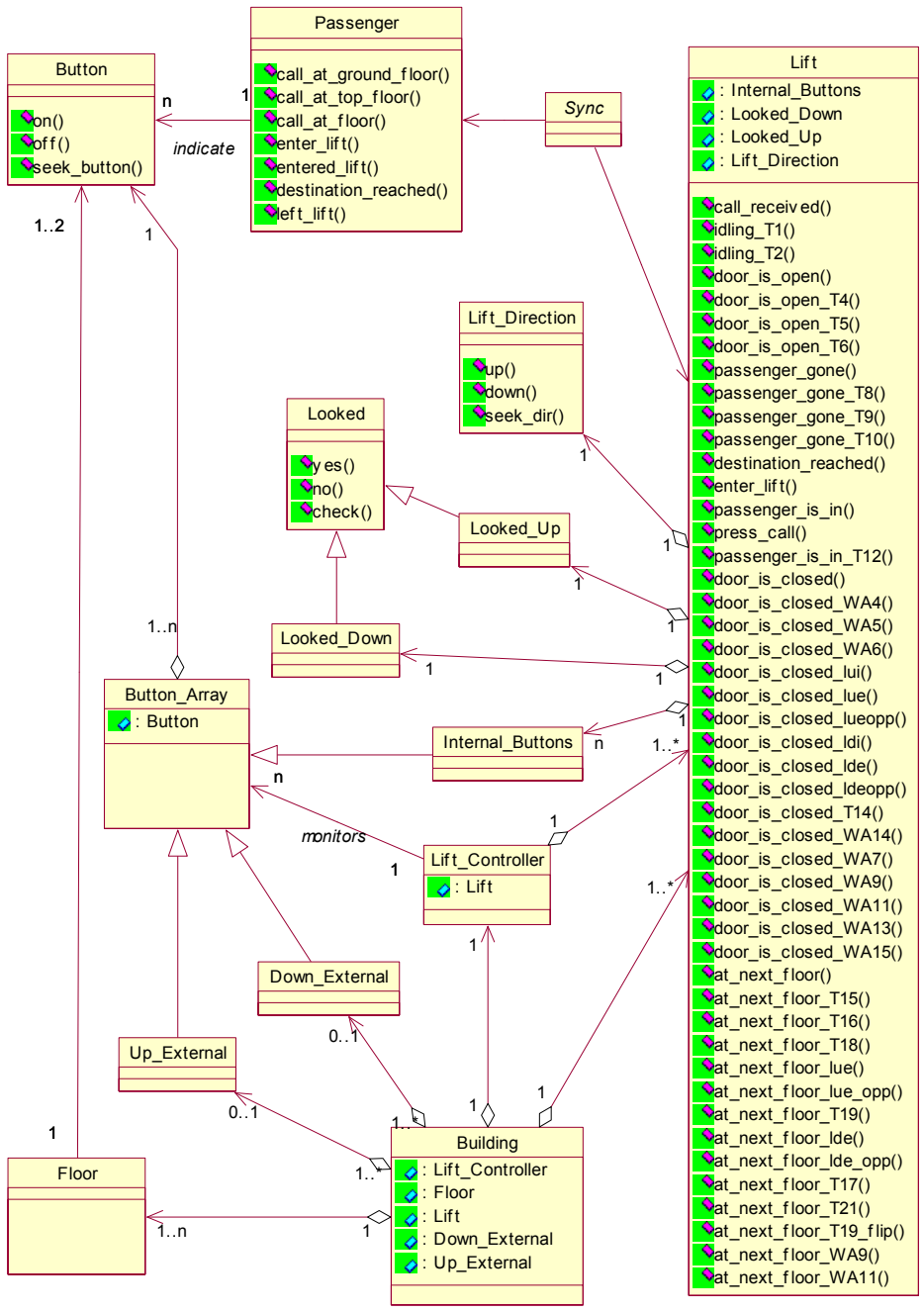

Figure 6 LIFT_SYSTEM_SAFE: Object model of the FSP model

Derived from the final 'corrected model', this object model depicts the structural and behavioural relationships between the FSP processes, superimposed over the pre-FSP object model. The superimposition provides contextualisation of the objects that were explored in FSP as processes.

In *Figure 5 Lift System: Lift object behavioural diagram* arrows represent transitions, derived from the developer model, as they are represented in the FSP model. Transitions that leave and enter the same state indicate recursive entry into that state.

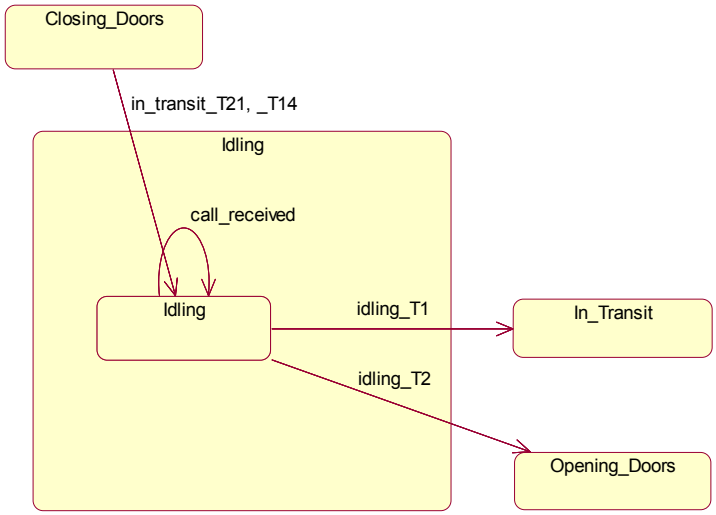

Figure 7 LIFT_SYSTEM_SAFE: IDLING Process

The idling state may only be reached by transiting from the Closing_Doors state. There are two transitions out of the idling state which are executable given satisfaction of the guards that apply to the event received – either a call from the current floor in which case the doors can open immediately, or a call from a floor other than the floor where the Lift is currently situated.

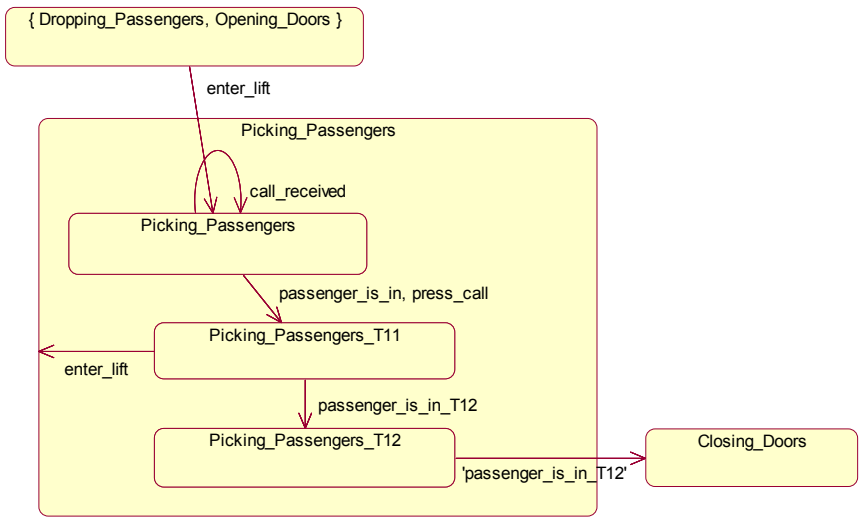

Figure 8 LIFT_SYSTEM_SAFE: PICKING_PASSENGERS Process

Picking_Passengers state is reachable from two states (Dropping_Passengers and Opening_Doors) via the same enter_lift event. While there are more passengers to pick up, the enter_lift event transits recursively to the Picking_Passengers state. Exit from this state is achieved once there are no more passengers to pick up – allowing the closing of the doors. Note that the developer model specifies passengers leave a lift before passengers enter a lift.

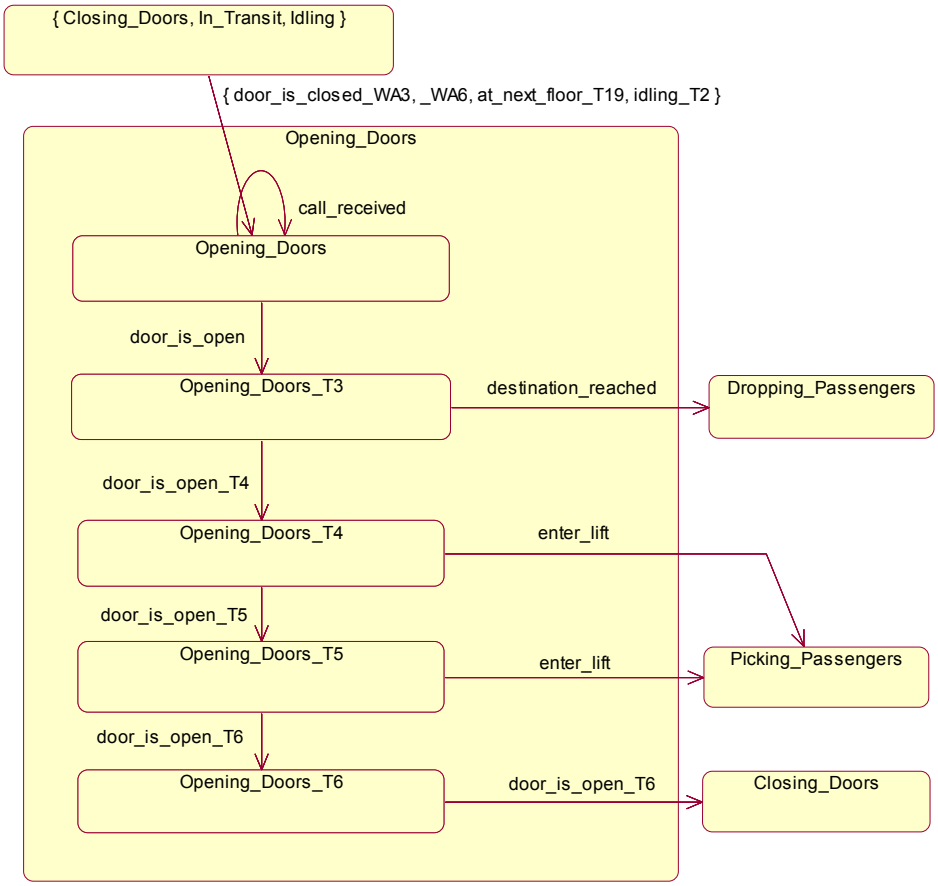

Figure 9 LIFT_SYSTEM_SAFE: OPENING_DOORS Process

There are three states that allow transit into the Opening_Doors state: Closing_Doors, In_Transit, and Idling. Likewise, there are three states to which the Opening_Doors state may exit to: Dropping_Passengers, Picking_Passengers and Closing_Doors.

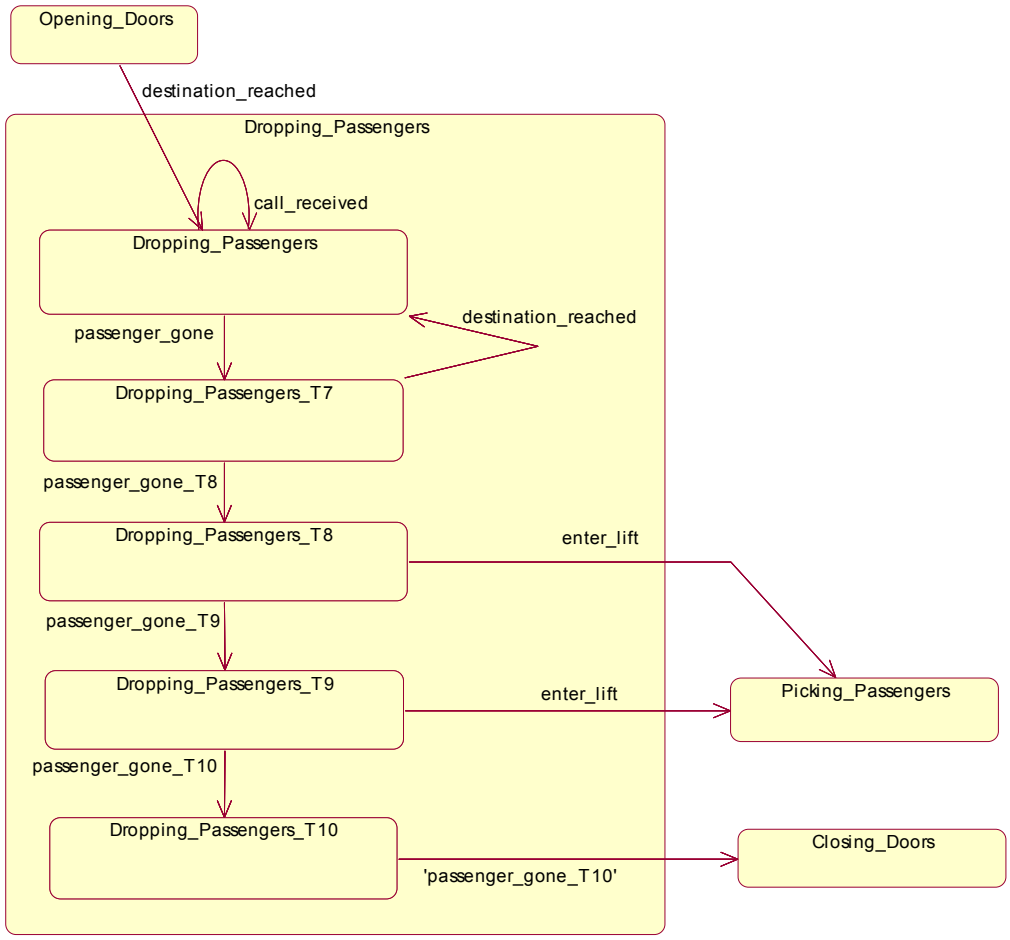

Figure 10 LIFT_SYSTEM_SAFE: DROPPING_PASSENGERS Process

Dropping_Passengers can only be reached from the Opening_Doors state, which produces a destination_reached event signifying an instruction to the passenger. If there is more than one passenger to depart at this floor, another destination_reached event is provided from within the Dropping_Passengers state. Once all passengers who are departing have departed, passengers from the current floor may enter the lift, or the lift will close its doors and proceed to execute the Closing_Doors state.

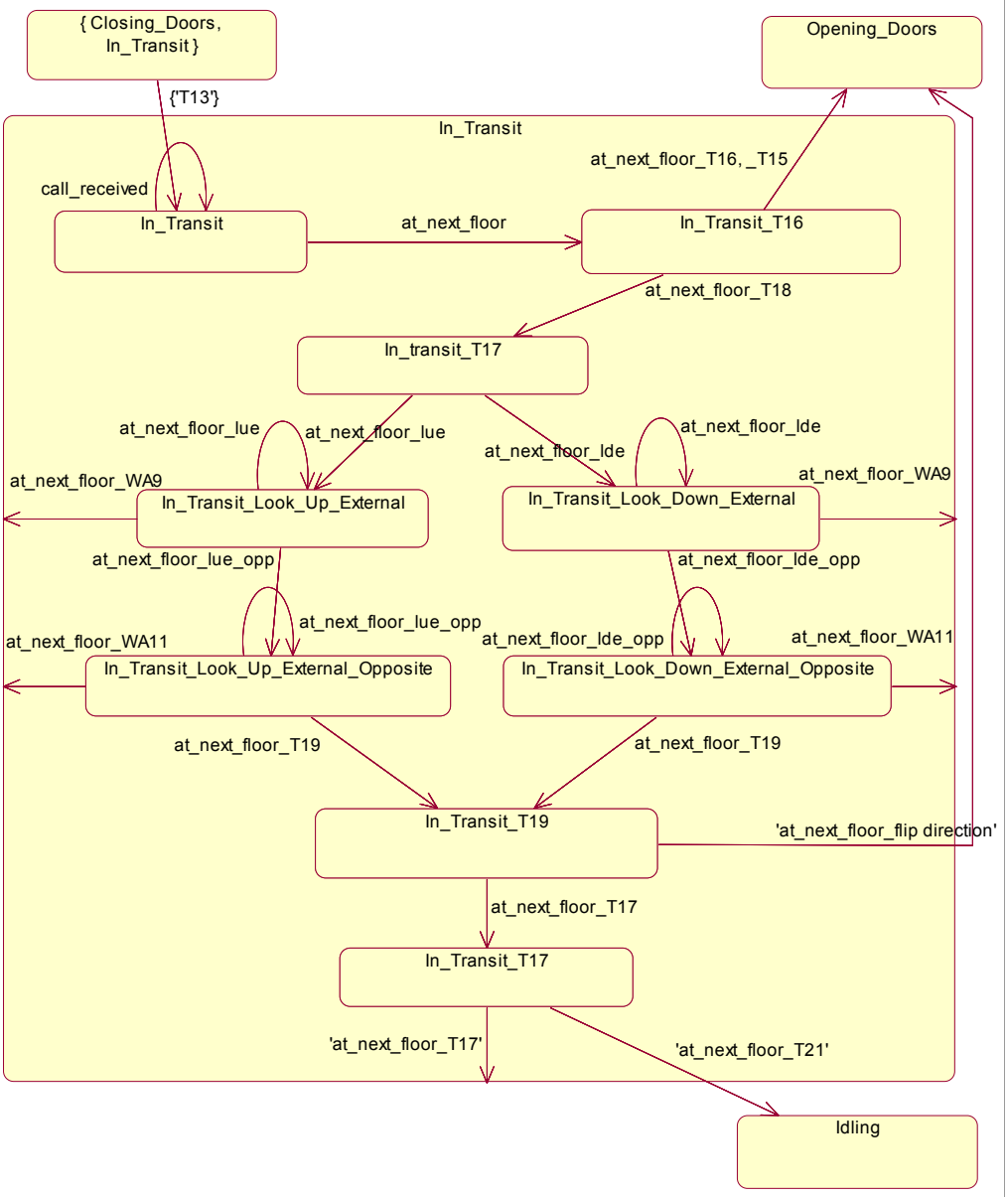

Figure 11 LIFT_SYSTEM_SAFE: IN_TRANSIT Process

In_Transit executes a shorter variant of the polling sequence termed the walk algorithm. Exit from this state is only achievable if there is a passenger to get into the lift at a floor, if there is a passenger to get out at a floor, or, no work to do which leads to the Idling state.

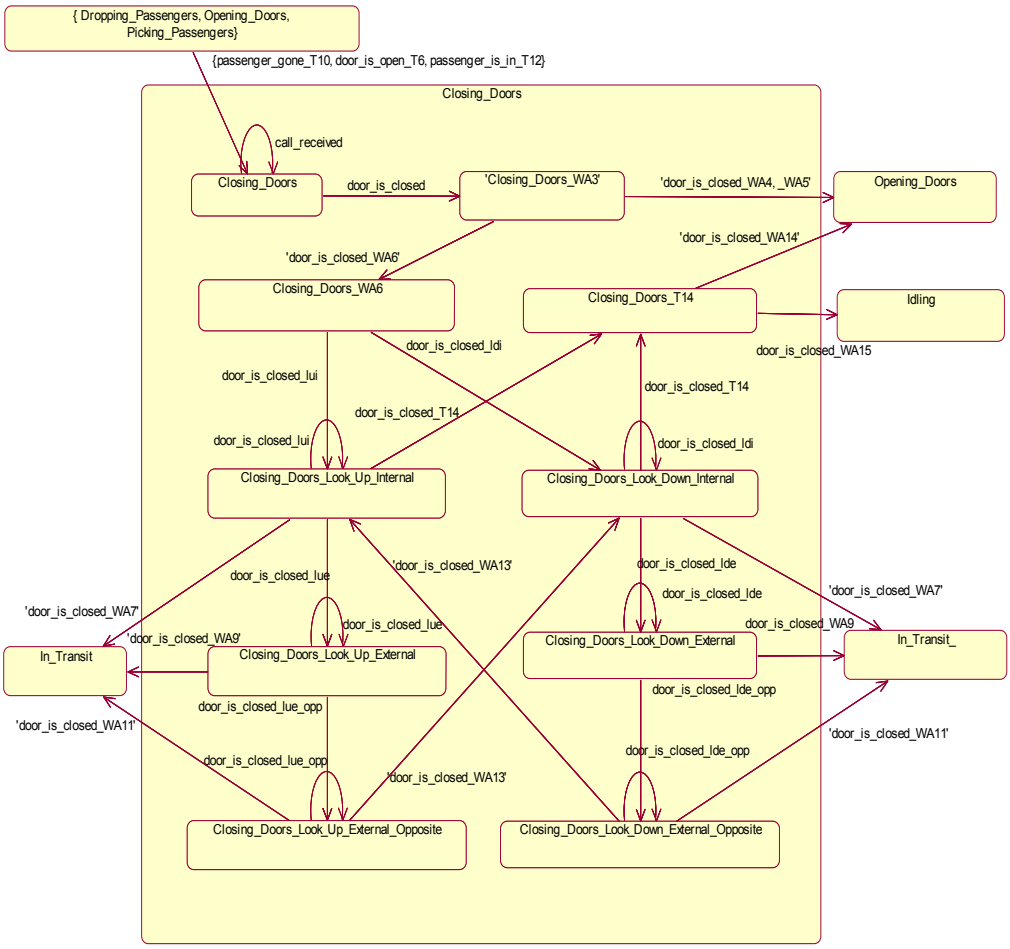

Figure 12 LIFT_SYSTEM_SAFE: CLOSING_DOORS Process

The Closing_Doors process is the busiest of all the processes in the Lift object. This process has the largest number of internal processes and also the greatest number of transitions out of the process into other processes. Closing_Doors encapsulates the Walk Algorithm, the core of which is repeated to a lesser extent in the In_Transit local process of the Lift object as shown in Figure 11 LIFT_SYSTEM_SAFE: In_Transit.

5.3.1 The Walk Algorithm

The *walk algorithm* is applied during the Closing_Doors state, a shorter variant is applied during the In_Transit state. The purpose of the *walk algorithm* is to resolve the direction of travel for the lift given the state of the various passengers in the system.

The algorithm can be traced in the corrected model listing, or in Figures 10 and 11, which depict the behavioural diagrams of the above-mentioned states. The following listing is drawn from the documentation within the FSP-model; it specifies a pseudo-code translation of the Walk Algorithm.

```
//-------------------------------------------------------------
//      WALK ALGORITHM
//      1.              We have to scan every floor for internal and
//                      external calls
//      2.              Start with the current floor,
//      3.              test the internal button for the current direction
//      4.              If true, OPEN_DOORS
//      5.              else, test external call for current direction
//      6.              test every floor internally in current lift
//                      direction until boundary floor, starting at the
//                      current floor (+-1).
//      7.              if true, IN_TRANSIT in that direction.
//      8.              else, when boundary reached, test in current
//                      direction for external calls for current direction
//                      (starting from current floor (+-1)).
//      9.              if any true, IN_TRANSIT in that direction
//      10.             else, when boundary reached, test in current
//                      direction for external calls for opposite direction
//                      (starting from current floor (+-1).
//      11.             if any true, IN_TRANSIT in the current direction
//      12.             else, flip lift direction,
//      13.             repeat from 5-12 (lift direction has been flipped
//      14.             If no calls at all then IDLING
//
//      NOTE 1: If there are no OCCUPANTS (p==0), internal
//              walk can be skipped.
//      NOTE 2: When testing external calls, guards must be
//              placed to block invalid floor/button combos,
//              i.e., a DOWN button on the BOTTOM_FLOOR
//      NOTE 3: Each Walk direction has a flag; when all flags
//              are set, the walk is complete, and we may progress
//              to IDLING. LOOKED flags only need to be set TRUE
//              in External walks, as they *ALWAYS* follow
//              Internal component of walk
//      NOTE 4: Steps 8 and 10 cannot be performed concurrently, the
//              end result is the quite the same, as someone going
//              in the direction who is further away from the lift
//              than someone who is closer but going in opposite
//              direction will face starvation
//-------------------------------------------------------------
```

Both the full *walk algorithm* and the shorter variant rely on a priority sequence for the order of polling. Polling commences internally, proceeds to the external buttons for the current direction, and then completes with the external buttons for the opposite direction - with respect to the lifts' current direction. Each poll seeks every button in the set, from the current floor to the boundary floor, by using a recursive process.

5.3.2 Implicit Transitions

Implicit Transitions are transitions denoted in the Implementation Specification behavioural diagrams by enclosing single quotes, eg 'at_next_floor_T21'. These transitions are differentiated from the normal, explicit transitions because they do not represent a transition to a sub-local process in the FSP-model, but to transitions from a section of code in the FSP-model to another section of code in the FSP-model within the same sub-local process, or, because the transition transits to a new local process without having a matching action statement in the FSP code.

5.4 Corrected Model to Developer Model

The final *corrected model* (LIFT_SYSTEM_SAFE, Section 11.6) is a result of iterative application of the FSP modelling process to an FSP-representation of the developer model. This section summarises the contributions of this process to the developer model, that is, solutions to errors - that were revealed by applying FSP – that have adjusted the developer model, *where the developer model was found to be in error*.

A corrected developer model in the format of Transition Tables is not offered as a task output from this process, as the corrected model serves the need for a specification upon which the Implementation Specification is built. Issues that arose during FSP analysis that have been documented have been presented in the previous sub-sections. This section mostly discusses the Lift object, and errors that impact the developer model representation of the Lift object.

5.4.1 Lift object errors revealed in Developer Model

The errors that inform the developer model are divided into three categories, those errors that reveal a fatal concurrency flaw in the developer model, those errors that reveal flaws of a non-fatal concurrency violation, and those errors that contribute to the developer model in filling out the specification for the final corrected model.

5.4.1.1 Fatal concurrency flaws in developer model

The developer model is found to contain a flaw in the specification of lift actions and the sequence in which the lift executes its polling for passengers to attend. When a passenger is at a floor and there are no other passengers, the developer model allows the lift to bypass checking at the floor it is currently at and go straight to polling all other floors, The polling subsequently ends, at the passengers floor, however the lift still does not check for a passenger going in the opposite direction to its current direction. Exposed through the generation of three separate errors in LTSA, (LiftAndPassengerPV3, LiftAndPassengerPV12, and LIFT_SYSTEM_D2), this is a fatal flaw in the model, that is, the event generated by the passenger calling a lift has not been correctly acted upon by the lift such that the lift halts and leaves the passenger stranded, almost ludicrously, as the lift is at the same floor as the passenger.

5.4.1.2 Non-fatal concurrency flaws in developer model

LiftAndPassengerPV17 exposes the situation where a number of people are alighting from the lift in a sequential order at the same floor. After each alighting, the Lift closes its doors and then opens them again, and then signals the passenger to alight. This is a non-fatal flaw as the passengers do still get to alight from the lift, just not in as timely a fashion as they would have if the door was to remain open between the deliveries of the alighting signal.

5.4.1.3 Contributory specifications

There are several errors that contribute to the correct specification of the *walk algorithm*; they are LiftAndPassengerPV4, LiftAndPassengerPV15, and LIFT_SYSTEM_PV3. The *walk algorithm* itself is not explicitly specified in the developer model, this is a product of the FSP modelling. The order for the polling of floors respective to the Lifts direction is drawn from the basic outline given implicitly in the developer model, but, as is described in Section 5.4.1.1 there are fatal flaws in that outline. Further testing of the explicit, expanded FSP *walk algorithm* contributes to the final correct specification.

5.5 Corrected Model to Client Description

Correcting the client description from the corrected model is, like the process of correcting the developer model, a continual feedback loop. In this thesis, this task was performed as one stage, rather than as an iterative, incremental contributor. The reason for this is that essentially, the refinement of the client description had already occurred in the process of building the developer model as detailed in Lakos & Malhotra (2002)!

There is still some input to the client description from the FSP modelling, particularly in the realm of contributory specifications. Firstly, the Walk Algorithm pseudo-code (Section 5.3.1) represents a prose form of the priority order of polling. Using this pseudo-code / prose we can generate the following text description:

Polling commences internally, proceeds to the external buttons for the current direction, and then completes with the external buttons for the opposite direction (with respect to the lifts' original current direction).
The original current direction is reversed at the boundary floors. Boundary floors are the top, bottom and current floors. Polling seeks from the current floor in the direction of the Walk Algorithms' current direction until the polling reaches a boundary floor.

There are three types of button to poll: internal, external up and external down. Neither up or down has greater priority, the direction in which to poll is dictated by the lifts current direction.
When the doors close the lift first checks to see if the internal button is lit, this is the last chance to intercept button events before departing (entering a travel state). Predicating on the first firing of a transition during the walk algorithm and treating the walk algorithm as atomic enables a priority polling scheme.

After testing the internal button (those conditions not being true), poll the external button for the current floor for the current direction. If no call there then

return to the internal buttons and poll every floor from the current floor plus one - in the direction of the poll - to the boundary floor (either the top or bottom floor).

If at any floor there is a call, exit the closing_doors state and travel in the current direction. There is no need to remember which floor has made the call as the re-polling is performed again in the in_transit state to decide when to cease travel and open the lift doors or to keep moving.

If there are no calls in the current direction for the current direction then test every floor in the current direction for people wanting to travel in the opposite direction to the lifts current direction, from the current floor plus one - in the current direction - to the boundary floor.
If there is a call then change state to in_transit, in that direction.

If there are still no calls reverse the lifts direction and repeat the process as just described from the point immediately after testing for the internal button for the current direction; so the polling is repeated from the second test but for the opposite to the current direction.

If this poll completes without firing a transition then there is no work to do so go to idling.

6 Implementation Details

This section brings together explanatory notes, concerns and comments upon the FSP model and also upon the process of FSP modelling in the context of this thesis. Section 6.1 outlines optimisations that are available in the corrected model to be performed. Section 6.2 clarifies the ordering of the production of deliverables. Section 6.3 refers to the specific version of the text description of the Lift Problem that is used in this thesis, while Section 6.4 describes some abstract concerns of modelling passengers. Section 6.5 explains the simple form of passive atomic functions used in the model, while Sections 6.6 and 6.7 are both concerned with different aspects of object inter-relationships.

6.1 Further Optimisations

An immediate optimisation that is possible in the FSP model, but that is destructive to the faithfulness of the corrected model to the developer model is to collapse and fold the FSP-model to its full extent.

This optimisation possibility arises from two situations, firstly where the state has been maintained to keep in step with the developer model. This treats the developer model as prescriptive to the FSP-model. To regard the FSP-model as prescriptive to the developer model takes FSP-analysis away from checking and analysis to directly informing the corrected model, and also directly informing the Implementation Specification. Further collapsing of state within the FSP-model is possible, as are morphologies of the actions of processes, and of the interaction of processes (synchronised transitions).

6.2 Dependencies

Dependencies in the FSP-analysis exist in the order of creation of Implementation Specification, with regard to the FSP modelling process. The Implementation Specification Object Model is dependent upon the finalised iterative process for

its construction and for the delineation of inter-relationships. Likewise, the Implementation Specification behavioural diagrams are dependent upon the finalised FSP-model for the complete, and correct, specification.

6.3 Derivation of Lift Problem Description

The initial description of the Lift Problem (Section 11.1) contains details that do not appear in the version of the Lift Problem that is used in this thesis as the initial text description. The first issue is the abstraction of the Lift Controller out of the developer model and subsequently the FSP model, and thus the corrected model.

The second issue is that timing is not considered significant to the extent that timing issues are removed from explicit treatment in the FSP-model.

6.3.1 Abstraction of Lift Controller

The original text description refers to a 'lift controller'; the controller is responsible for receiving external button events and dispatching these calls to the 'best placed' lift. To simplify the problem domain, and to maintain correspondence with the original developer model, only a single lift is modelled throughout the process as presented. With only a single lift, the lift controller becomes superfluous - when the lift is idle, it can receive the external button event directly, and when it is not idle, the external button events will be queued by the button-structure at the appropriate floor.

6.3.2 Timing Issues

The original problem description refers to the significance of the time domain for the problem solution. Given as relevant considerations for timing are delays for opening and closing doors, and the amount of time it takes for a passenger to enter or leave a lift. Timing concerns are less pronounced as a necessary consideration for the FSP-modelling. FSP performs an action in unit time; synchronous actions are also performed in unit time. So while the duration of the action is not relevant to the model, the ordering of the actions is still of prime importance.

6.4 Passengers

Passenger objects exhibit a well-defined, uncomplicated lifecycle. Their lifecycle does not possess internal cycles; instead a passenger object follows an ordered series of steps with clear and unambiguous start and end points. The 'well-behaved passenger' notion defines a set of behaviours exhibited by the passenger object on which the FSP-model is built.

6.4.1 Well Behaved Passengers

Well-behaved passengers have some behaviour that is helpful to the model. Firstly, a passenger is not making a contract when they indicate their desired direction of travel; the passenger object may get out at the same floor, a floor in the direction of their indicated direction of travel, or a floor in the opposite direction of their indication. This constraint was not placed on the model to impose 'real-world ' conditions as much as it was placed to lessen the size of the state structures carried by a lift through its traversal of its processes.

6.5 Atomic Walks

Atomic Walk is the term used in this thesis to delimit the sequence of actions and state changes that exist between two passenger synchronised actions. In this model this can be a sequence the length of the *walk algorithm* (section 5.3.1).

During the iterative modelling stage it became apparent that some form of mutual exclusion was needed, so that a sequence of actions, like a subset of the *walk algorithm*, could complete as if it were one action, allowing the safe interleaving of the *walk algorithm* and any passenger object actions. The problem partly solved itself as a solution as found in the FSP-relabelling mechanism, which allows synchronisation of events between objects. Also, tightening of the atomic walk sequences, and the development of the *walk algorithm*, allow 'instantaneous' polling of the external buttons.

6.6 Tight Coupling of FSP Processes

An issue with the synchronisation mechanism in FSP (relabelling) is that it binds otherwise independent processes. Binding through agents (buttons) allows looser associations between exclusive objects. Looser associations contribute to cleaner mappings to UML entities, i.e. object-oriented objects.

6.7 Separation of Components in UML Model

Relaxing of the coupling between Lift and Passenger processes in the model risks the concurrency properties that have been proven. Definition of the shared interface between Lift and Passenger in the UML Object Model is not complete as the correct OO abstraction for the objects' synchronisation is still at issue.

7 Research Discussion

This section presents overall findings in respect to the question: "To what extent is FSP useful for - and to the development of - formally validated and verified models?"

7.1 Summary of Methodology

An initial text description forms the input specification of a problem domain that we are to map to a solution domain via the twin paths of validation-refinement and verification of concurrent processes.

The initial text description is transformed to the developer model via refinement, and the developer model is transformed to an analysis model via finite state process language that is formally verified by the analysis of the reachable state-space of the growing solution model.

This growing model evolves into the corrected model, having had the semantic concepts of the problem domain mapped into the formal space of the ‘provably correct’. Iteration of the verification and refinement procedures proceeds until defined exit conditions are met, the tightest exit condition being the non-generation of formal errors.

Once a corrected model is released from the iterative process, it is transformed to a Unified Modelling Language Specification that may be a candidate for Implementation; otherwise it must undergo further validation and verification cycles.

7.2 The Implementation Specification and pre-FSP Object Models

Section 7.1 stated that ‘The pre-FSP object model served as the basis for the extraction of the entities from the initial text description.’ These entities were subsequently mapped into the FSP model as local processes. Composition of the

local processes produced the corrected model. From the corrected model is derived the Implementation Specification.

Another use for the pre-FSP object model (Section 3.5) is to compare it to the object model produced from the corrected model (Section 5.2). The pre-FSP object model is acknowledgeably incomplete; it does not purport to be regarded as complete beyond being a possible structure for the system. This explains the lack of operations or attributes in the pre-FSP model.

However, the Implementation Specification object model does contain numerous operations, but is only suggestive as regards attributes. Aggregation relationships immediately suggest containment structures of objects as attributes. Beyond these implicit attributes (that are present in the model due to aggregation) are *implementation specific* data-types and data-structures. These attributes have not been introduced into the Implementation Specification.

7.3 State Space and Computational Limits

One large concern with Finite State modelling is the size of the total reachable states in any one system. The LIFT_SYSTEM_SAFE model generates 1267 states, 1327 transitions between those states, and requires ~5.5 megabytes to explore in memory. These numbers are for when the model is loaded with one passenger and three floors.

Tables 3 - 6 (Section 11.6) contain the data for the number of states, the number of transitions, the memory that was required, and the potential state space for varying numbers of Passengers and Floors. The fact that the model is tested for a bounded number of Lifts and Passengers proves the correctness of the concurrent processes *to the tested limits*, but no further. We may infer safe operation of the model for numbers beyond the exhaustively-checked range, but we cannot claim that a model that has been tested to a finite limit is *provably correct* beyond the tested limit.

Mitigation of state-space size may be possible by separating the abstraction process of a system into staggered stages.

As a component (or set of components) is modelled and found correct, we may be able to slide it away ‘from under the microscope’ to then focus on a connected component. Proving correctness over a chain of ‘windowed abstractions’ would also be prone to computational expense, and so require the interfaces between components not only be correct, but to also be abstracted away from state variables to minimise the state-space.

7.4 Well-Behaved Passengers and other Constraints

The well-behaved passenger constraint (Section 6.4.1) allows the reduction of state variables in the model, by disconnecting the passengers indicated floor of travel from the actual floor of travel. This also serves the purpose of defining one behavioural characteristic that we may expect of real-world passengers, i.e. unreliability.

There are many other facets of real-world behaviour that are not included within the *well-behaved* description. For example, the passenger who presses every one of the Lift’s internal buttons causing the Lift to stop at every floor regardless of actual passengers in the lift desiring to travel to those floors has not been modelled.

So, without providing a specification for *any-imaginable* behaviour, and instead modelling a finite set of behaviours, we may only claim a provably correct model for a constrained passenger type.

A further constraint on the thesis research process, that did however provide the framework for the FSP-modelling, was the use of the developer model as a specification. A disadvantage of adopting the developer model as the template for the FSP model is a concomitant short-circuiting of the design process, removing the need in this thesis to build up the developer model from the client description.

This affects the refining of the client description from the analysis process by removing the need to draw on the client description (and hence the client) for information. Thus the feedback element of the refining process becomes stilted and 'one-way'.

8 Conclusion

Section 3.6 defines the two major tasks of this thesis, to recap they are:

1 *apply FSP to verification of behavioural analysis*,

2 *apply iterative–FSP to the validation-led methodology*.

The first task has been attempted on the model of the Lift System, and has been found to be effective for revealing the presence of behavioural anomalies in both fatal and non-fatal categories.

The fatal error category covers any errors in the model that halt the system, or that cause the system's integrity to be compromised, that is, undefined in respect of the semantic concepts that specify allowable and desired behaviours from the system. Fatal errors (as reported by LTSA) were found in the developer model and they were resolved (again, to the satisfaction of LTSA).

The non-fatal error category covers errors that do not halt the system, do not compromise the integrity of the system, but do expose redundant or unnecessary behaviours. While not 'dangerous' in the same way that fatal errors are, non-fatal errors detract from the efficiency of a system.

The resolution of a non-fatal error in this thesis (Section 5.4.1.2) frees the Lift of pointless opening and closing of doors. It is within the scope of the constraints (Section 7.4) to add that this semantic definition of 'pointless behaviour' applies within the world mapped out by the model, in a real-world model where Passenger behaviour was much more closely defined we may wish the lift to close it's doors after a given delay.

The second task was not purposefully applied during the FSP-modelling process partly due to the concentration upon the developer model, but mostly, because of the correctness of the initial text description. The initial text description had already undergone validation-led refinement, so perhaps this result was to be expected, however the corrected model adds to the initial description in other ways. The generation of Contributory Specifications provides substantial information about the system under scrutiny. These Contributory Specifications are transformable from their initial state (either FSP or pseudo-code prose) to text descriptions of the anticipated behavioural characteristics of the system.

Having mapped back to the world of the client we are in a position to seek *validation* of the *proposed solution* - in a language closer to that of the client than are many of the numerous models we are accustomed to quite normally using in the development and implementation of software systems.

9 Further Work

9.1 Optimisations on Implementation Specification

The Implementation Specification may be further optimised. The state-chart diagrams for lift-states Picking Passengers, Dropping Passengers, and Opening Doors (Figures 8, 10, 9 respectively) share a common characteristic: the last sub-state in each of these states offers no choice; rather the sub-state represents a set of sequential instructions to be followed by a transition to a new state. The representation of the set of instructions as a sub-state in the state-chart diagrams is maintained as an artefact of the FSP model. Optimisation of the Implementation Specification can be achieved by the removal of the sub-state in each diagram, necessitating the inclusion of the set of instructions in the previous state in each diagram.

9.2 Development of Architectural Description

Given the basis of FSP in the Darwin Architectural Modelling Language (Section 2.1.7), it may be instructive to translate the FSP model of the Lift System into the Darwin notation, i.e. present the *corrected model* as a component model. Addressing the high-level concerns of component-modelling may provide insight into a formal solution for the model of shared interactions, as discussed in Section 6.7.

10 References

Bass L., Clements P. & Kazman R., *Software Architecture in Practice*, Reading, Massachusetts: Addison-Wesley, 1998

Booch G., *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin Cummings, Redwood City. 1993

Booch G., Rumbaugh J. & Jacobson I., *The Unified Modelling Language User Guide*, Addison-Wesley 1998

Brinksma, E., *'What is the Method in Formal Methods?'* in Formal Description Techniques, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '91, editors K. R. Parker & G. A. Rose, IFIP Transactions, North-Holland, 1992

Dijkstra E. W., *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, Communications of the ACM August 1975 Vol 18 Num 8.

Gamma E., Helm R., Johnson R. & Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995

Giannakopoulou D., Magee J. & Kramer J., *Fairness and Priority in Progress Property Analysis*, Department of Computing, Imperial College of Science, Technology and Medicine, Research Report, DoC 99/3, December 1999,
URL: citeseer.nj.nec.com/giannakopoulou99fairnes.html

Hoare C. A. R., *Communicating Sequential Processes*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1985

Holzmann G. J., *Designing Executable Abstractions*, Proceedings of Workshop on Formal Methods in Software Practice, Clearwater Beach, FL, March 1998. ACM Press 1998

Jacobson I., Christersson M., Jonsson P., & Overgaard G., *Object-Oriented Software Engineering*, Addison-Wesley, Menlo Park, CA, USA 1992

Lakos C. A. & Malhotra V. M., *Validation Led Development of Software Specifications*, International Journal of Modelling and Simulation ACTA Press,
Volume 22, Number 1, 2002, pp. 57-74

Magee J., *Behavioural Analysis of Software Architectures using LTSA*, Proceedings of the 21st International Conference on Software Engineering (ICSE '99), pp634-637, 1999, Los Angeles, CA, ACM 1999

Magee J., Dulay N., Eisenbach S., & Kramer J., *Specifying Distributed Architectures*, Proc. 5th European Software Engineering Conference (ESEC 95), Springer-Verlag, Berlin, Vol 989, pp 137-153, 1995

Magee J. & Kramer J., *Concurrency: State Models and Java Programs*, John Wiley and Sons. 1999

Magee J., Pryce N., Giannakopoulou D. & Kramer J., *Graphical Animation of Behavior* (sic) *Models*, Proceedings of the 22nd international conference on Software engineering, p.499-508, June 04-11, 2000, Limerick, Ireland, ACM 2000

Medvidovic N., Rosenblum D., Redmiles D. & Robbins J., *Modelling Software Architectures in the Unified Modelling Language*, ACM Transactions on Software Engineering and Methodology, Vol 11, No 1, January 2002, pp2-57.

Milner R., *A Calculus of Communicating Systems*, volume 92 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1980

Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam, *Object Constraint Language Specification* version 1.1, 1 September 1997
URL:http://www3.ibm.com/software/ad/library/standards/ad970808_UML11_OCL.pdf

Rumbaugh J., Blaha M., Premerlani W., Eddy F. & Lorensen W., *Object-Oriented Modelling and Design*, Prentice Hall, 1991

Shaw M. & Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Alan Apt, Prentice Hall 1996

Sommerville I., *Software Engineering*, Fifth Edition, Addison-Wesley Publishing Company, 1995

11 Appendix

11.1 Original Case Study - The Lift Problem

A building is serviced by several identical lifts. These lifts travel between the floors of the building and are controlled by a common controller. Each floor, with the exception of the ground and the top floor, has two call buttons - one for each direction of the travel. For obvious reasons, the ground and the top floor have only one button.

Passengers arriving at a floor press the call button appropriate to their direction of travel. They wait in a first-in-first-out (FIFO) queue for their turn to enter a lift. When a lift arrives and the doors open, the current passengers destined for this floor get out, and then as many waiting passengers as possible get in. These embarking passengers press appropriate buttons for their destinations. When the lift arrives at their destination, they get out.

Each call for a lift is registered by the controller. For each call of a lift, the controller determines if an idle lift is better placed than the currently active lifts to service the call. If this is the case, the idle lift is dispatched towards the calling floor. If several idle lifts are equally placed to service the call, one is chosen at random. Otherwise, an active lift will eventually arrive at the calling floor and will be able to service the call.

An idle lift continues to be inactive and stationary at a floor, with the door closed, till it is activated by the controller in response to a call from a newly arrived passenger. An idle lift, when activated, begins to travel towards the calling floor and serves the other passengers like other active lifts. An active lift continues to move upwards and downwards serving the waiting passengers till it finds no further remaining work. At this stage it becomes idle. A lift going upwards halts at various floors to drop passengers and to pick up new passengers. It does not change its direction of travel until it reaches a floor where it is no longer carrying a passenger, and where there are no waiting passengers on any floor above. The lift may change direction in order to service the passengers waiting to travel in the other direction. If this also yields no further work for the lift, the lift enters its idle state. An analogous behaviour is shown by a lift travelling downwards.

We assume that timing is significant for the problem solution. Thus, each lift travels at a fixed rate and needs some fixed delay to open and close its door. Likewise, the passengers take a fixed delay to enter and exit a lift. Each lift has as many destination buttons as there are floors in the building.

11.2 Validation-Led Specifications

Transition	Start State	Event	Guard	Actions	Next State
P1	Enters the system	Passenger	True	Presses the call button appropriate for the intended direction of travel	Waiting for a lift
P2	Waiting for a lift	EnterLift	True	Schedule EnteredLift event to be delivered after necessary delay	Entering a lift
P3	Entering a lift	EnteredLift	True	Press the destination button of the lift and send PassengerIsIn event to the lift	Waiting for the lift to reach destination
P4	Waiting for the lift to reach destination	Destination _Reached	True	Schedule LeftLift event to be delivered after necessary delay	Leaving the lift
P5	Leaving the lift	LeftLift	True	Send PassengerGone event to the lift and Passenger	Leaves the system

Table 1 Transition Descriptions for the Lifecycle Passenger (Lakos & Malhotra 2002 p63).

Transition	Start State	Event	Guard	Actions	Next State
T1	Idling	CallRec eived	Call from a floor other than where the lift is idling	Determine the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit

T2	Idling	CallReceived	Call from the floor where the lift is idling	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door
---	Any state other than Idling	CallReceived	True	Empty	No change
T3	Opening Door	DoorIsOpen	Someone in the lift wants to get out on this floor	Send DestinationReached event to a passenger in the lift for this destination	Dropping Passengers
T4	Opening Door	DoorIsOpen	No one in the lift wants to go out here and the lift is not full and there is a waiting passenger on this floor to go in the current direction of lifts travel	Send EnterLift event to a waiting passenger going in the current direction of the lift's travel	Picking Passengers
T5	Opening door	DoorIsOpen	No one left in the lift and no pending call requires lift to continue to travel in its current direction; there is, however, a passenger on this floor waiting to travel in the other direction	Reverse the direction of travel and send EnterLift event to a waiting passenger going in the new direction of the lift's travel	Picking Passengers
T6	Opening Door	DoorIsOpen	None of the guards in T3, T4, and T5 is true.	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door
T7	Dropping Passengers	PassengerGone	Someone in the lift wants to get out on this floor	Send DestinationReached event to a passenger in the lift for this destination	Dropping passengers

T8	Dropping Passengers	PassengerGone	No one in the lift wants to get out here and the lift is not full and there is a waiting passenger on this floor to go in the current direction of lift's travel	Send EnterLift event to a waiting passenger going in the current direction of the lift's travel	Picking passengers
T9	Dropping Passengers	PassengerGone	No one left in the lift and no pending call requires lift to continue to travel in its current direction; there is, however, a passenger on this floor waiting to travel in the other direction	Reverse the direction of travel and send EnterLift event to a waiting passenger going in the new direction of the lift's travel	Picking passengers
T10	Dropping Passengers	PassengerGone	None of the guards in T7, T8, and T9 is true	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door
T11	Picking Passengers	PassengerIsIn	The lift is not full and there is a waiting passenger on this floor to go in the current direction of the lift's travel	Send EnterLift event to a waiting passenger	Picking Passengers
T12	Picking Passengers	PassengerIsIn	The lift is full or there is no waiting passenger on this floor to go in the current direction of the lift's travel	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door

T13	Closing door	DoorIs Closed	The lift is not empty or there is call from a passenger that the lift should attend	Determine the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit
T14	Closing door	DoorIs Closed	The lift is empty and there is no call that the lift need attend	Empty	Idling
T15	In transit	AtNext Floor	Someone on this floor is waiting to travel in the direction of lift's travel	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door
T16	In transit	AtNext Floor	Someone in the lift wants to get out on this floor	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door
T17	In transit	AtNext Floor	The lift is not empty and no one in the lift wants to get out here and there is no waiting passenger on this floor to go in the current direction of lifts travel	Schedule AtNextFloor event to be delivered after necessary delay	In transit
T18	In transit	AtNext Floor	No one in the lift wants to get out here and there is no waiting passenger on this floor to go in the current direction of lift's travel and there are waiting passengers on the floors in the current direction of lift's travel	Schedule AtNextFloor event to be delivered after necessary delay	In transit

T19	In transit	AtNext Floor	Guards for T15, T16, T17, and T18 are all false and there is a waiting passenger on this floor to go in the direction opposite to the current direction of lift's travel	Reverse the direction of the travel and schedule DoorIsOpen event to be delivered after necessary delay	Opening door
T20	In transit	AtNext Floor	Guards for T15, T16, T17, T18, and T19 are all false and there are waiting passengers on the floors in the direction opposite to the current direction of lift's travel	Reverse the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit
T21	In transit	AtNext Floor	Guards for T15, T16, T17, T18, T19, and T20 are all false	Empty	Idling.

Table 2 Transition Descriptions for the Lifecycle Lift (Lakos & Malhotra 2002 p67).

11.3 Early stage FSP Listing for a Lift (Section 5.1.3)

```
/*
Lift.lts
Direct translation of Transition Table Three
State variables: boolean:DIRECTION, boolean:BOOL
Assumption: Passengers are well-behaved*/

const UP = 1
const DOWN = 0
range DIRECTION = DOWN..UP

const TRUE = 1
const FALSE = 0
range BOOL = FALSE..TRUE

const N = 2
range FLOOR = 1..N

const MAX_OCCUPANTS = 1
range OCCUPANTS = 0..MAX_OCCUPANTS

const MAX_QUEUE = N-1
range QUEUE_INDEX = 0..MAX_QUEUE

set S = {[FLOOR],[FLOOR][FLOOR]}
```

```
PORT = (send[f:FLOOR] -> PORT[f]),
PORT[f:FLOOR] = (send[f2:FLOOR] -> PORT[f2][f]
                    |receive[f] -> PORT),
PORT[f2:FLOOR][f:FLOOR]    = (receive[f] -> PORT[f2]).



LIFT = IDLING[1][0][UP],

IDLING[f:FLOOR][p:OCCUPANTS][d:DIRECTION] =
       (call_received[call:FLOOR][dir:DIRECTION] ->
       /*     T2           call from the current floor */
       (when (f==call) delay -> OPENING_DOOR[p][f][dir]
       /*     T1           call from a floor below */
       |when (f>call && f!=0) delay -> IN_TRANSIT[p][f-1][call][DOWN]
       /*     T1           call from a floor above */
       |when (f<call && f!=N) delay -> IN_TRANSIT[p][f+1][call][UP])),

OPENING_DOOR[p:OCCUPANTS][f:FLOOR][d:DIRECTION] = (door_is_open[f][d] ->
       /*     T3           someone wants to get out here */
              (when (p>0) destination_reached[f] ->
DROPPING_PASSENGERS[p][f][d]
       /*     T4           no one wants to get out at this floor
                           AND lift is not full
                           AND waiting passenger on floor going in lifts
direction
                           **ENTRY POINT STEP 1**     */
              |when (TRUE && p<MAX_OCCUPANTS && TRUE) enter_lift[204] ->
PICKING_PASSENGERS[p][f][d]
       /*     T5           no one left in the lift
                           AND no pending call in current direction
                           AND passenger on this floor for other
direction */
              |when (p==0 && TRUE && TRUE) enter_lift[205] ->
PICKING_PASSENGERS[p][f][d]
       /*     T6           !( T3 | T4 | T5 ) */
              |delay -> CLOSING_DOOR[p][f][d])
       /*     T-           True */
              |call_received[call:FLOOR][d2:DIRECTION] ->
OPENING_DOOR[p][f][d]),

DROPPING_PASSENGERS[p:OCCUPANTS][f:FLOOR][d:DIRECTION] = (passenger_gone
->
       /*     T7           someone else wants to get out here (we have
not moved) */
       (when (p>0 && TRUE) delay -> destination_reached[f] ->
DROPPING_PASSENGERS[p-1][f][d]
       /*     T8           no one wants to get out at this floor
                           AND lift is not full
                           AND there is a waiting passenger on floor for
current direction */
       |when (TRUE && p<MAX_OCCUPANTS && TRUE) enter_lift[308] ->
PICKING_PASSENGERS[p+1][f][d]
       /*     T9           no one left in lift
                           AND no pending call for current direction
                           AND passengers at floor for other direction */
       |when (p==0 && TRUE && TRUE) enter_lift[309] ->
PICKING_PASSENGERS[p+1][f][d]
       /*     T10          !(T7 | T8 | T9) */
       |when (p==0) delay -> CLOSING_DOOR[p][f][d])
       /*     T-           True */
       |call_received[call:FLOOR][d2:DIRECTION] ->
DROPPING_PASSENGERS[p][f][d]),

PICKING_PASSENGERS[p:OCCUPANTS][f:FLOOR][d:DIRECTION] =
(passenger_is_in[dest:FLOOR][dest>f] ->
       /*     T11          lift is not full
```

```
                        AND there is a waiting passenger on this floor
for current direction
                        **ENTRY POINT STEP 2**     */
            (when (p<MAX_OCCUPANTS && TRUE) enter_lift[411] ->
PICKING_PASSENGERS[p+1][f][d]
      /*    T12           lift is full
                        OR there is no waiting passenger on this floor
for current direction */
            |when (p==MAX_OCCUPANTS || TRUE) delay ->
CLOSING_DOOR[p][f][d])
      /*    T-            True */
            | call_received[call:FLOOR][d2:DIRECTION] ->
PICKING_PASSENGERS[p][f][d]),

CLOSING_DOOR[p:OCCUPANTS][f:FLOOR][d:DIRECTION] = (door_is_closed ->
      /*    T13           lift is not empty
                        OR call from passenger to attend (must
determine direction) */
            (when (p>0 || TRUE ) delay -> IN_TRANSIT[p][f][f][d]
      /*    T14           lift is empty
                        AND no calls to attend */
            |when (p==0 && TRUE) delay -> IDLING[f][p][d])
      /*    T-            True */
            | call_received[call:FLOOR][d2:DIRECTION] ->
CLOSING_DOOR[p][f][d]),

IN_TRANSIT[p:OCCUPANTS][f:FLOOR][call:FLOOR][d:DIRECTION] =
(at_next_floor ->
      /*    T15           there is a waiting passenger at this floor for
current direction */
      (when (TRUE) delay -> OPENING_DOOR[p][f][d]
      /*    T16           someone wants to get out here */
      |when (TRUE) delay -> OPENING_DOOR[p][f][d]
      /*    T17           lift is not empty
                        AND no one wants to get out here
                        AND no waiting passenger on this floor for
current direction */
      |when (p>0 && TRUE && TRUE) delay -> IN_TRANSIT[p][f][call][d]
      /*    T18           no one in lift wants to get out here
                        AND no waiting passenger on this floor for
current direction
                        AND there are waiting passengers on floors in
direction of travel */
      |when (TRUE && TRUE && TRUE) delay -> IN_TRANSIT[p][f][call][d]
      /*   T19            !( T15 | T16 | T17 | T18 )
                        AND there is a waiting passenger on this floor
for opposite direction of
                        current travel */
      |when (p==0) delay -> OPENING_DOOR[p][f][d]
      /*   T20            !( T15 | T16 | T17 | T18 | T19 )
                        AND there are waiting passengers on other
floors in opposite direction
                        for opposite direction of travel */
      |when (p==0) delay -> IN_TRANSIT[p][f][call][d]
      /*    T21    !( T15 | T16 | T17 | T18 | T19 | T20 ) */
      |when (p==0) delay -> IDLING[f][p][d])
      |call_received[call][d2:DIRECTION] -> IN_TRANSIT[p][f][call][d]).


||A_LIFT = (LIFT || up:PORT || down:PORT )
            /{
            call_received[f:FLOOR][UP]/up.send[f],
            call_received[f:FLOOR][DOWN]/down.send[f],
            passenger_is_in[dest:FLOOR][UP]/up.send[dest],
            passenger_is_in[dest:FLOOR][DOWN]/down.send[dest],
            door_is_open[f:FLOOR][UP]/up.receive[f],
            door_is_open[f:FLOOR][DOWN]/down.receive[f]
            }.
```

11.4 Early FSP Listing Passenger

```
/*
Passenger
Direct translation from Table 1
No State variables
*/


PASSENGER = (arrival -> ENTERS_SYSTEM),

ENTERS_SYSTEM = (passenger -> PRESS_CALL
              |passenger -> WAITING_FOR_LIFT),

PRESS_CALL = (call_direction -> WAITING_FOR_LIFT),

WAITING_FOR_LIFT = (enter_lift ->ENTERING_LIFT
              |wait -> WAITING_FOR_LIFT),

ENTERING_LIFT = (entered_lift -> PRESS_BUTTON
              |passengerIsIn -> WAIT_IN_LIFT),

PRESS_BUTTON = (passengerIsIn -> WAIT_IN_LIFT),

WAIT_IN_LIFT = (wait -> WAIT_IN_LIFT
              |destination_reached -> LEAVE_LIFT),

LEAVE_LIFT = (wait -> LEAVE_LIFT
              |left_lift -> LEAVES_SYSTEM),

LEAVES_SYSTEM = (passenger_gone -> PASSENGER).
```

11.5 Error Documentation Summary

Iterative FSP-analysis: Error Documentation Summary.

Name	Specification	Development	Model File	Trace Divergence/ Halt	Description of Progress Violation or Deadlock	Comments/Solution
ProgressViolation	No	Yes	n/a	n/a	ProgressViolation caused by typographic error	replaced an 'e' with an 'a'
ProgressViolation2.txt	No	Yes	LiftAndPassengerPV2.lts	593	Passenger arrives at floor three and calls lift. Lift arrives at floor three and passenger gets in. Presses for floor 1. The door closes, and lift starts walk. Walk proceeds to external down and then goes into idling.	Problem arises from a prior process in the walk, specifically the first guard of LOOK_DOWN_INTERNAL required range check to be equal to lowest floor number to allow the recursive call to reach the internal button at the first floor.
ProgressViolation3.txt	Yes	Yes	LiftAndPassengerPV3.lts	477	Passenger arrives at floor 3 and calls lift. The lift arrives and the passenger enters the lift. The passenger presses to get out at floor 3 (the same floor). The lift commences walk and proceeds to idling.	This problem occurs with the implementation of the specification and the interpretation of that. See PV 6. This problem is not addressed immediately, in favour of addressing the Deadlock3 condition. This problem is deferred, and then arises again, albeit with a slightly different walk path.
Deadlock3.txt	No	Yes	LiftAndPassengerD3.lts	507	Passenger arrives at floor 3 and calls lift. The lift arrives and opens its door. The passenger enters and presses to go to floor 1. The lift closes its doors and travels to the first floor where it then opens its doors and the passenger gets out. Then a passenger arrives at the second floor and presses the call button (Note that the passenger process to the call point is interleaved with the lift process). The lift travels to the second floor but doesn't stop to let the passenger in, instead it keeps travelling to the third floor where it opens its door to let the passenger in, who is still on the second floor!	As the passenger is wishing to travel down from the second floor, the call button that is triggered is the external down button. The lift looks up first for buttons registered that are for the up direction. As there are none for the second floor, the walk proceeds to look at the third floor. The process definition for the third floor walk in this context performs an immediate seek on all external buttons, reagrdless of the floor. This creates an error condition as we cannot have an up button on the top floor. The false up button is registered (rather than unregistered, ie it is in the 1 position), so the lift believes it should open the doors here.
ProgressViolation4.txt	No	Yes	LiftAndPassengerPV4.lts	478	Passenger arrives at top floor, and calls lift. The lift travels up and then reaches the top floor. The door opens and the passenger gets in and presses the button to get out at this same floor.The door closes and the lift starts its walk, and then enters the idle state. The passenger is still in the lift.	The lifts walk is unsuccessful because the instruction at line 478 assumed that the conditions at that point meant the entire walk had been performed, regardless of direction walk was performed (up or down). Replacing the instruction to go to IDLE with an instruction to continue the walk but in the opposite direction allows the lift to look up and find the call to the top floor.

ProgressViolation5.txt	No	Yes	LiftAndPassengerPV5.lts	385	Passenger arrives at top floor and presses the call button. The lift travels up and then the lift reaches the top floor. The lift opens its door and the passenger enters the lift. The passenger indicates to travel to the current floor. The door closes and then the lift does **not** perform a walk but halts.	Enabling a guard after closing the door and before embarking on the walk allows the internal departure at the current floor to be found and then acted upon by the inclusion of an instruction to open the door at line 395.
Deadlock6.txt	No	Yes	LiftAndPassengerD6.lts	283	Passenger arives at the ground floor and presses the call button. The door opens and the passenger enters the lift and indicates to travel to the current floor. The door closes and then opens and the passenger leaves the lift. A passenger then arrives at the ground floor but does not press call yet. The lift begins it walk but halts.	The actions that return the number of calls in any queue are called at the top of the DROPPING_PASSENGERS process without excluding illogical combinations of floor and button direction. For example, the bottom floor in this scenario has a down button. The lift reads this error state as an instruction to travel down, but cannot, so halts. To rectify this situation the initial seeks in DROPPING_PASSENGERS are wrapped up in local proceses so that the correct behaviour is applied to each of the three seperate cases. This creates the processes DP_ANY_FLOOR, DP_TOP_FLOOR and DP_BOTTOM_FLOOR
ProgressViolation7.txt	No	Yes	LiftAndPassengerPV7.lts	508	Passenger arrives at floor 3 and presses the down button. The lift travels up and opens the door. The passenger gets in and presses the button for floor 3. The door closes and then reopens. The passenger gets out and another passenger arrives at the same floor. The passenger does not get into the lift, the lift starts to walk the queues but runs into a terminal set at pauser_e_d.	The lift enters it walk activity and enters an infinite loop. Altering line 496 to have a transition out to an IDLING state allows the walk to terminate.
Deadlock8.txt	No	Yes	LiftAndPassengerD8.lts	309	The passenger arrives at the middle floor, calls lift to travel up. The lift travels up and then the passenger enters the lift (even though the door did not open) , and indicates to get out at this floor. The door closes and then the door reopens, and the passenger leaves the lift. Then a passenger arrives at the first floor and the door closes before the passenger makes a call. The lift finishes its post closing door walk and then the passenger presses the call button on the ground floor. The lift travels to the ground floor and then the passenger enters the lift (the door is again closed). The passenger indicates to travel to this floor, then the door closes, reopens and the pasenger leaves the lift. A paasenger arrives at the same floor (ground floor) and the lift goes into its walk and halts.	This deadlock arises from incorrect button usage within the lift local process IN_TRANSIT. This implementation allows the bottom floor to have a down button. There is another issue revelaed in this trace, and that is that the lift can currently go directly to a floor and pick up a passenger without opening its doors! This issue is handled later, at this point the incorrect button possibility is handled. Introduction of a three way branch into the IN_TRANSIT process allows the illogical combinations of floor and buttons to be excluded. The new code block is found at lines 554 - 590.

ProgressViolation9.txt	No	Yes	LiftAndPassengerPV9.lts	496	The passenger arrives at the top floor and calls the lift. The lift rises one floor and then closes its doors - even though they should be closed at this time anyway. The walk through the queues also does not accept the call from the top floor but instead terminates.	Correction of the typographical error in the process calls between lines 568-586 and also correcting the transition and process calls between lines 595 - 606. The first set of corrections were required due to a development error, as is the second error (which was alluded to in the description of Deadlock8). The second case corrects the situation where the doors do not need to be open for the passenger to get in to the lift.
ProgressViolation10.txt	No	Yes	LiftAndPassengerPV10.lts	315	The passenger arrives at the second floor and calls the lift. The lift travels up and the door opens. The passenger enters the lift at floor 2 and indicates to travel up to floor 3. The door closes and then the lift completes the walk. The lift is then in the idling state, and the passenger is stuck in the lift.	The lift should have found the departure indicator for the top floor, but was not able to reach this check because the guard in the recursive call blocked any seeks to the top floor. Essentially a development error, but is comparable with the external recursive fuctions that require the tightness of the similar guard. Resolved by tightening the guard to the action that performs the button seek. Line 450
ProgressViolation11.txt	No	Yes	LiftAndPassengerPV11.lts	184	The Passenger arrives at the second floor and calls the lift. The lift travels up and then opens the door at the second floor. The passenger enters the lift, and indicates to travel up to the top floor. The lift travels up to the top floor and opens the door. The passenger gets out and then a passenger arrives at the third floor. The lift door is closed before the passenger can press a call button. After the lift finishes its walk the passenger presses the call button and the door opens. The passenger enters the lift and indicates to travel to the current floor. The door is closed and then the door is opened. The passenger leaves the lift and then a passenger arrives at the same floor. The walk halts in an errror state.	Correcting the guard at line 184 stops the walk sequence from entering an error state. The issue was that a button seek was being performed on the up button for the top floor - which does not exist! The guard is tightened by adding a check against seeking the up button on the top floor.
ProgressViolation12.txt	Yes	Yes	LiftAndPassengerPV12.lts	417	The passenger arrives at the second floor and then calls the lift. The lift travels up and then the door opens. The passenger enters the lift and indicates to travel to current floor. The door closes and then the door opens. The passenger leaves and then a passenger arrives. The door is closed and the lift performs a walk. The passenger calls the lift and the door opens, then the walk halts.	This violation occurs when there is a passenger at a floor wanting to go in the direction opposte to the lifts current direction. The lift will check all floors for its current direction first, so a seek will not be fired if the lift is opposite to the passenger. This is fixed by adding an extra process called BRANCH_OD . Because the Lift is empty the Passenger shoud be able to use the Lift.

ProgressViolation13.txt	Yes	Yes	LiftAndPassengerPV14.lts	540	The passenger arrives at the second floor and calls the lift. The lift then travels up to the second floor and opens the door. The passenger enters the lift and indicates to travel up to the top floor. The door is closed and the walk completes and the lift enters the idle state. The passenger is still in the lift!	The passenger had initially indicated to travel down, however when the passenger was in the lift the passenger indicated to travel up. The lift though is expecting to go down and so performs the walk in the downward direction. When the lift walk reaches the bottom it believes that it has looked everywhere and so goes to idle. The fix for this involved adding in two new local processes at the composite level and using them as boolean masks against having seeked up or down, so that the lift would look up if still needed and vice versa. These new local processes are termed LOOKED.
ProgressViolation 14.txt	Yes	Yes	LiftAndPassengerPV15.lts	n/a	SAFE, for one passenger, one lift, three floors and one call point (IDLING).	No fixes required. However, call point additions to major states of lift is next task.
ProgressViolation15.txt	Yes	Yes	LiftAndPassengerPV16.lts	552	Passenger arrives at the second floor and presses call. The lift reaches the second floor and the door opens. The Passenger enters the lift, and indicates to travel to current floor. The door closes and then the door opens and the Passenger leaves the lift. A Passenger arrives at the third floor and indicates to travel to the current floor. The door closes and the lift enters the walk. The walk completes, the lift goes to idle and the passenger is still in the lift.	The walk appears to correctly execute, with the final checks finishing the wlak and going to idling. This error is due to a boundary flip situation, where th e walk gets to the top of the floors looking for up buttons (because the lift is travelling up), doesn't see that there is a button down (external) at the top floor. Solved by the addition of an extra clause in the walk, to account for boundary flip.
ProgressViolation16.txt	Yes	Yes	LiftAndPassengerPV17.lts	457	Passenger arrives at floor 2, indicates to travel up. The lift travels up and the door opens. The Passenger enters the lift, indicates to travel to the current floor. The door closes and then the door opens. The Passenger leaves, and then a Passenger arrives at this floor, and indicates to travel up. The lift enters its walk, terminates, and the passenger is stuck at the floor.	Need to add branching to closing door to allow for safe check of external buttons at any floor, before going for walk.
ProgressViolation17.txt	Yes	Yes	LiftAndPassengerPV18.lts	442	Passenger arrives at first floor, calls lift. Second Passenger arrives at first floor and calls lift. The doors open and the first passenger enters the lift, indicates to travel up to the second floor. The second passenger enters the lift and indicates to travel to the current floor. The door closes, then reopens. The second passenger gets out and a third passenger arrives at floor 3. THen we have a premature halt with the first passenger stuck in the lift.	The door should not have closed when there was someone in the lift to get out at the floor where the doors were open already.

ProgressViolation18.txt	No	Yes	LIFT_SYSTEM.lts	511	Passenger 1 arrives at floor 1, passenger 2 arrives at floor 2. Passenger 1 calls lift, passenger 2 calls lift. The lift arrives at floor 2, then does a walk, then the dorrs open. Passenger 2 enters the lift, and indicates to travel up to floor 3, the door closes. The the lift reaches the ground floor and the walk terminates, with a passenger in the lift.	The lift should have travelled up after reaching the second floor and picking up the passenger, as that was the direction it was headed in. This error led to a rewite of the program, same structure but all freshly written.
LIFT_SYSTEM_PV_.txt	No	Yes	LIFT_SYSTEM.lts	125	empty trace, terminal state or T20 and T21 and assocoated actions. (These are the flip bits)	Occurs because call_received is mapped to lift in IDLE state only, meaning that certain conditions can never occur, ie , a lift will never have to do the 'flip' when a passenger can only call during the idle state.
LIFT_SYSTEM_PV1.lts	No	Yes	LIFT_SYSTEM_D_2.lts	110	Passenger calls lift and then lift gets stuck in progress violation.	Logical error in the placement of call_received in every major state. Also issues with incrementing floor numbers between different processes of the walk
LIFT_SYSTEM_D_2.txt	Yes		LIFT_SYSTEM_D_2.lts		P calls to go up from 2nd floor, lift goes to second floor, 2nd passenger calls lift from third floor, to go down. The 1st P gets in and indicates to go down! The lift does it scan, is still in a mood to go up, and sees the third floor call. The lift goes to the third floor, and should open the door, instead, hits T17 which fulfills current conditions against guards, and so travels on to the fourth floor without stopping at the third.	This problem would not arise if the Ps did what they said what they were going to, ie travel in the direction they indicated. In the absence of a property to enforce this behaviour, which is not realistic anyhow, another solution is needed. What should happen? The lift should pick up the 2nd passenger at the third floor, then move on from there.
LIFT_SYSTEM_PV3.lts			LIFT_SYSTEM_SAFE.lts		Passenger calls at second floor, lift arrives at second floor, second passenger calls at second floor, then lift opens door. First Passenger enters lift and presses third floor button, second passenger enters lift and presses first floor button, then door closes. Passenger three calls at the third floor. Lift travels to third floor and opens door, then first passenger leaves lift, door closes and passenger one calls at second floor. Trace halts. Passenger three is left at third floor, Passenger one at second floor, and passenger two (going to first floor) is in lift.	Provision of correct reentrant behaviour to the walk algorithm corrects this behaviour

11.6 State-Space Data

	Floors							
Passengers	3	4	5	6	7	8	9	10
1	1267	2748	5067	8416	12987	18972	26563	35952
2	8236	24765	57899	116066	209454	*350011*	*551445*	*829224*
3	56664	234756	*697580*	*1692762*	*3580800*			
4	*405172*							
5	*2966788*							

Table 3 State-Space Data: Reachable States.

	Floors							
Passengers	3	4	5	6	7	8	9	10
1	1327	2868	5267	8716	13407	19532	27283	36852
2	9146	27197	62949	125118	224180	*372371*	*583687*	*873884*
3	68388	278076	*812408*	*1943094*	*4060332*			
4	*537502*							
5	*4337180*							

Table 4 Sate Space Data: Transitions.

	Floors							
Passengers	3	4	5	6	7	8	9	10
1	5517	5640	6197	6786	6795	7989	8982	10037
2	5710	737227	798245	914163	1143147	*672888*	*565285*	*737403*
3	14974	893856	*738346*	*788071*	*1010344*			
4	*62585*							
5	*437926*							

Table 5 State Space Data: Memory Used.

	Floors							
Passengers	3	4	5	6	7	8	9	10
1	2.00E+24	2.00E+29	2.00E+33	2.00E+36	2.00E+40	2.00E+44	2.00E+47	2.00E+50
2	2.00E+36	2.00E+45	2.00E+52	2.00E+58	2.00E+67	2.00E+73	2.00E+79	2.00E+86
3	2.00E+41	2.00E+51	2.00E+58	2.00E+64	2.00E+73			
4	2.00E+53							
5	2.00E+57							

Table 6 State Space Data: Potential State Space.

11.7 LIFT_SYSTEM_SAFE.lts

```
const NUM_PASSENGERS = 1
const MAX_PASSENGERS = 1
const NUM_FLOORS = 3
const BOTTOM_FLOOR = 1
const TOP_FLOOR = NUM_FLOORS

const DOWN = 0
const UP = 1

const FALSE = 0
const TRUE = 1

range BOOL = FALSE..TRUE
range DIRECTION = DOWN..UP
range FLOOR = 1..NUM_FLOORS
range OCCUPANTS = 0..NUM_PASSENGERS
//             MIDDLE_FLOORS is in ERROR if NUM_FLOORS<=2
range MIDDLE_FLOORS = 2..NUM_FLOORS-1

//-------------------------------------------------------------------------------
//                             PASSENGER
//-------------------------------------------------------------------------------

PASSENGER =
(
               call_at_ground_floor ->       WAITING_FOR_LIFT[1]
       |       call_at_top_floor -> WAITING_FOR_LIFT[NUM_FLOORS]
       |       call_at_floor[f:MIDDLE_FLOORS][d:DIRECTION] -> WAITING_FOR_LIFT[f]
),


WAITING_FOR_LIFT[f:FLOOR] =
(
       enter_lift[f] -> ENTERING_LIFT[f]
),

ENTERING_LIFT[f:FLOOR] =
(
       entered_lift[f] -> press_destination[g:FLOOR] -> WAIT_IN_LIFT[g]
),

WAIT_IN_LIFT[g:FLOOR] =
(
       destination_reached[g] -> LEAVE_LIFT[g]
),

LEAVE_LIFT[g:FLOOR] =
(
       left_lift[g] -> PASSENGER
).



//-------------------------------------------------------------------------------
//                             BUTTON
//-------------------------------------------------------------------------------


BUTTON = BUTTON[0],

BUTTON[i:0..NUM_PASSENGERS] =
(
               when (i<NUM_PASSENGERS) on -> BUTTON[i+1]
       |       when (i>0) off -> BUTTON[i-1]
       |       seek_button[i] -> BUTTON[i]
).
```

```
//-------------------------------------------------------------------------------
//                          LIFT_DIRECTION
//-------------------------------------------------------------------------------


LIFT_DIRECTION = LIFT_DIRECTION[UP],

LIFT_DIRECTION[d:DIRECTION] =
(
                up -> LIFT_DIRECTION[UP]
        |       down -> LIFT_DIRECTION[DOWN]
        |       seek_dir[d] -> LIFT_DIRECTION[d]
).



//-------------------------------------------------------------------------------
//                          LOOKED
//-------------------------------------------------------------------------------

LOOKED = LOOKED[FALSE],

LOOKED[b:BOOL] =
(
                yes -> LOOKED[TRUE]
        |       no -> LOOKED[FALSE]
        |       check[b] -> LOOKED[b]
).



//-------------------------------------------------------------------------------
//                          LIFT
//-------------------------------------------------------------------------------



LIFT( LIFT_DIR = 'lift_direction,
                DPT_COUNT = 'dptCount,
                BTN_UP = 'btnUp,
                BTN_DOWN = 'btnDown,
                LOOKED_UP = 'looked_up,
                LOOKED_DOWN = 'looked_down
                ) = IDLING[1],

IDLING[f:FLOOR] =
(

                //      NOTE:   g is destination floor (origin of call to be synced
                //                      with lift_controller), not lift location
                call_received[g:FLOOR] ->
                //              T1              call from a floor below
                if (g<f && f>BOTTOM_FLOOR) then
                (
                        idling_T1 -> [LIFT_DIR].down -> IN_TRANSIT[0][f-1]
                )
                else
                //              T1              call from a floor above
                if (g>f && f<NUM_FLOORS) then
                (
                        idling_T1 -> [LIFT_DIR].up -> IN_TRANSIT[0][f+1]
                )
                else
                //              T2              call from the current floor
                if (g==f) then
                (
                        idling_T2 -> OPENING_DOORS[0][f]
                )

),

OPENING_DOORS[p:OCCUPANTS][f:FLOOR] =
(
                call_received[g:FLOOR] -> OPENING_DOORS[p][f]
```

```
                |
                door_is_open[f] ->
                [DPT_COUNT][f].seek_button[j:0..NUM_PASSENGERS] ->
                [LIFT_DIR].seek_dir[d:DIRECTION] ->

                //              T3      someone wants to get out here
                //                      j>0     departure counter
                //                      p>0     guard against error
                if (j>0 && p>0) then
                (
                        destination_reached[f] ->
                        DROPPING_PASSENGERS[p-1][f]
                )
                else
                        OPENING_DOORS_T4[p][f]
),


OPENING_DOORS_T4[p:OCCUPANTS][f:FLOOR] =
(
        door_is_open_T4 ->
        [DPT_COUNT][f].seek_button[j:0..NUM_PASSENGERS] ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->


        //              T4              no one wants to get out at this floor
        //                              (j==0)
        //                      AND     lift is not full
        //                              (p<MAX_PASSENGERS)
        //                      AND     waiting passenger on floor going
        //                              in lifts direction
        //                              (f<NUM_FLOORS) guard against an error state

        if (j==0 && p<MAX_PASSENGERS && d==UP && f<NUM_FLOORS) then
        (
                [BTN_UP][f].seek_button[i:0..NUM_PASSENGERS] ->
                if (i>0) then
                (
                        [BTN_UP][f].off ->
                        enter_lift[f] ->
                        PICKING_PASSENGERS[p+1][f]
                )
                else
                //      other transitions may be valid
                        OPENING_DOORS_T5[p][f]
        )

        //      (switch on d)
        else
        if (j==0 && p<MAX_PASSENGERS && d==DOWN && f>BOTTOM_FLOOR) then
        (
                [BTN_DOWN][f].seek_button[k:0..NUM_PASSENGERS] ->
                if (k>0) then
                (
                        [BTN_DOWN][f].off ->
                        enter_lift[f] ->
                        PICKING_PASSENGERS[p+1][f]
                )
                else
                //      other transitions may be valid
                        OPENING_DOORS_T5[p][f]
        )
        else
        //      test next transition
                OPENING_DOORS_T5[p][f]
),


OPENING_DOORS_T5[p:OCCUPANTS][f:FLOOR] =
(
        door_is_open_T5 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T5      no one left in the lift
        //              (p==0)
```

```
        //              AND no pending call in current direction
        //              (implied by T4 not firing)
        //              AND passenger on this floor for other direction
        //                              (must test, if none then T6)
        //                              Reverse Lift Direction

        if (p==0) then
                if (d==UP && f>BOTTOM_FLOOR) then
                (
                        [BTN_DOWN][f].seek_button[i:0..NUM_PASSENGERS] ->
                        if (i>0) then
                        (
                                [BTN_DOWN][f].off ->
                                [LIFT_DIR].down ->
                                enter_lift[f] -> PICKING_PASSENGERS[p+1][f]
                        )
                        else
                                OPENING_DOORS_T6[p][f]
                )
                else
                // d==DOWN because d!=UP
                if (f<NUM_FLOORS) then
                (
                        [BTN_UP][f].seek_button[l:0..NUM_PASSENGERS] ->
                        if (l>0) then
                        (
                                [BTN_UP][f].off ->
                                [LIFT_DIR].up ->
                                enter_lift[f] -> PICKING_PASSENGERS[p+1][f]
                        )
                        else
                                OPENING_DOORS_T6[p][f]
                )
                else
                        OPENING_DOORS_T6[p][f]
        else
                OPENING_DOORS_T6[p][f]
),


OPENING_DOORS_T6[p:OCCUPANTS][f:FLOOR] =
(
        //              T6              !( T3 | T4 | T5 )
        door_is_open_T6 -> CLOSING_DOORS[p][f]
),


DROPPING_PASSENGERS[p:OCCUPANTS][f:FLOOR] =
(
        call_received[g:FLOOR] -> DROPPING_PASSENGERS[p][f]

        |
        passenger_gone[f] ->
        [DPT_COUNT][f].seek_button[j:0..MAX_PASSENGERS] ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T7      someone else wants to get out here (we have not moved)
        //                              (j>0)
        //                              (p>0)   test against error
        if (j>0 && p>0) then
        (
                destination_reached[f] ->
                DROPPING_PASSENGERS[p-1][f]
        )

        else

        //      T8              no one wants to get out at this floor
        //                      (j==0)
        //                      AND lift is not full
        //                      (p<MAX_PASSENGERS)
        //                      AND there is a waiting passenger on floor for
        //                      current direction
        //                      (switch to test)
        if (j==0 && p<MAX_PASSENGERS && d==UP && f<NUM_FLOORS) then
        (
```

```
                [BTN_UP][f].seek_button[i:0..NUM_PASSENGERS] ->
                if (i>0) then
                (
                        [BTN_UP][f].off ->
                        enter_lift[f] ->
                        PICKING_PASSENGERS[p+1][f]
                )
                else
                //      other transitions may be valid
                        DROPPING_PASSENGERS_T9[p][f]
        )
        //      (switch on d)
        else
        if (j==0 && p<MAX_PASSENGERS && d==DOWN && f>BOTTOM_FLOOR) then
        (
                [BTN_DOWN][f].seek_button[k:0..NUM_PASSENGERS] ->
                if (k>0) then
                (
                        [BTN_DOWN][f].off ->
                        enter_lift[f] ->
                        PICKING_PASSENGERS[p+1][f]
                )
                else
                //      other transitions may be valid
                        DROPPING_PASSENGERS_T9[p][f]
        )
        else
                DROPPING_PASSENGERS_T9[p][f]
),


DROPPING_PASSENGERS_T9[p:OCCUPANTS][f:FLOOR] =
(
        passenger_gone_T9 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T9              no one left in lift
        //                      (p==0)
        //                      AND     no pending call for current direction
        //                      implied by reaching this point
        //                      AND     passenger at floor for other direction
        //                      (test now)

                        if (d==UP && f>BOTTOM_FLOOR && p<MAX_PASSENGERS) then
                        (
                                [BTN_DOWN][f].seek_button[i:0..NUM_PASSENGERS] ->
                                if (i>0) then
                                (
                                        [BTN_DOWN][f].off ->
                                        [LIFT_DIR].down ->
                                        enter_lift[f] -> PICKING_PASSENGERS[p+1][f]
                                )
                                else
                                        DROPPING_PASSENGERS_T10[p][f]
                        )
                        else
                        // d==DOWN because d!=UP
                        if (f<NUM_FLOORS && p<MAX_PASSENGERS ) then
                        (
                                [BTN_UP][f].seek_button[j:0..NUM_PASSENGERS] ->
                                if (j>0) then
                                (
                                        [BTN_UP][f].off ->
                                        [LIFT_DIR].up ->
                                        enter_lift[f] -> PICKING_PASSENGERS[p+1][f]
                                )
                                else
                                        DROPPING_PASSENGERS_T10[p][f]
                        )
                        else
                                DROPPING_PASSENGERS_T10[p][f]

),

DROPPING_PASSENGERS_T10[p:OCCUPANTS][f:FLOOR] =
(
```

```
        passenger_gone_T10 -> CLOSING_DOORS[p][f]
),

PICKING_PASSENGERS[p:OCCUPANTS][f:FLOOR] =
//      NOTE:   The Passenger is already in the lift...
(
        call_received[g:FLOOR] -> PICKING_PASSENGERS[p][f]

        |
        passenger_is_in[f] ->
        press_call[destination:FLOOR] ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T11             lift is not full
        //                      (p<MAX_PASSENGERS)
        //                      AND there is a waiting passenger on this
        //                      floor for current direction
        //                      (switch on direction to test)

        if(p<MAX_PASSENGERS) then

                if (d==UP && f<NUM_FLOORS) then
                (
                        [BTN_UP][f].seek_button[i:0..NUM_PASSENGERS] ->
                        if (i>0) then
                        (
                                [BTN_UP][f].off ->
                                enter_lift[f] ->
                                PICKING_PASSENGERS[p+1][f]
                        )
                        else
                        //      other transitions may be valid
                                PICKING_PASSENGERS_T12[p][f]
                )
                //      (switch on d)
                else
                if (d==DOWN && f>BOTTOM_FLOOR) then
                (
                        [BTN_DOWN][f].seek_button[k:0..NUM_PASSENGERS] ->
                        if (k>0) then
                        (
                                [BTN_DOWN][f].off ->
                                enter_lift[f] ->
                                PICKING_PASSENGERS[p+1][f]
                        )
                        else
                        //      other transitions may be valid
                                PICKING_PASSENGERS_T12[p][f]
                )
                else
                        PICKING_PASSENGERS_T12[p][f]
        else
                PICKING_PASSENGERS_T12[p][f]

),


PICKING_PASSENGERS_T12[p:OCCUPANTS][f:FLOOR] =
(
        picking_passengers_T12 ->

        //      T12             lift is full
        //                      (p==MAX_PASSENGERS)
        //                      OR there is no waiting passenger on this floor for
        //                      current direction
        //                      (implied by reaching this point - last test(T12))

        //      NOTE:   This whole process (T12) could be handled in the
        //              previous process. It is retained as a separate process
        //              here for clarity.

        if(p==MAX_PASSENGERS) then
                CLOSING_DOORS[p][f]
        else
                CLOSING_DOORS[p][f]
),
```

```
CLOSING_DOORS[p:OCCUPANTS][f:FLOOR] =
(
        call_received[g:FLOOR] -> CLOSING_DOORS[p][f]

        |
        door_is_closed ->
        [DPT_COUNT][f].seek_button[i:0..MAX_PASSENGERS] ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T13     lift is not empty
        //              (p!=0)
        //      OR      call from passenger to attend
        //              (must determine direction - use walk algorithm)
        //      BUT     Must check this floor first
        //              internally (for departures) and
        //              externally (for current direction)

        //-------------------------------------------------------------
        //      WALK ALGORITHM
        //      1.              We have to scan every floor for internal and
        //                      external calls
        //      2.              Start with the current floor,
        //      3.              test the internal button for the current direction
        //      4.              If true, OPEN_DOORS
        //      5.              else, test external call for current direction
        //      6.              test every floor internally in current lift
        //                      direction until boundary floor, starting at the
        //                      current floor (+-1).
        //      7.              if true, IN_TRANSIT in that direction.
        //      8.              else, when boundary reached, test in current
        //                      direction for external calls for current direction
        //                      (starting from current floor (+-1)).
        //      9.              if any true, IN_TRANSIT in that direction
        //      10.             else, when boundary reached, test in current
        //                      direction for external calls for opposite direction
        //                      (starting from current floor (+-1).
        //      11.             if any true, IN_TRANSIT in the current direction
        //      12.             else, flip lift direction,
        //      13.             repeat from 5-12 (lift direction has been flipped
        //      14.             If no calls at all then IDLING
        //      15
        //
        //      NOTE 1:         If there are no OCCUPANTS (p==0), internal
        //                      walk can be skipped.
        //      NOTE 2:         When testing external calls, guards must be
        //                      placed to block invalid floor/button combos,
        //                      ie, a DOWN button on the BOTTOM_FLOOR
        //      NOTE 3:         Each Walk direction has a flag; when all flags
        //                      are set, the walk is complete, and we may progress
        //                      to IDLING. LOOKED flags only need to be set TRUE
        //                      in External walks, as they *ALWAYS* follow
        //                      Internal component of walk
        //      NOTE 4:         Steps 8 and 10 cannot be performed concurrently, the
        //                      end result is the quite the same, as someone going
        //                      in the direction who is further away from the lift
        //                      than someone who is closer but going in opposite
        //                      direction will face starvation
        //-------------------------------------------------------------


        //      WA_3.           Test for internal calls at this floor
        //                      (i>0)
        if (p!=0 && i>0) then
                //              WA_4
                //              Is an incorrect spec?
                //              Should it be PP?
                OPENING_DOORS[p][f]

        else

        //                      WA_5.   Test for external calls at this floor,
        //                                      for the current direction.
```

```
        //                                      (d==UP)
        //                                      guard (WA_NOTE2) (f<TOP_FLOOR)

        if(d==UP && f<TOP_FLOOR) then
        (
                [BTN_UP][f].seek_button[j:0..NUM_PASSENGERS] ->
                if (j>0) then
                        OPENING_DOORS[p][f]
                else
                        CLOSING_DOORS_WA6[p][f]
        )
        else
        //                      WA_5.   Test for external calls at this floor,
        //                                      for the current direction.
        //                                      (d==DOWN) switch on direction
        //                                      guard (WA_NOTE2) (f>BOTTOM_FLOOR)

        if (d==DOWN && f>BOTTOM_FLOOR) then
        (
                [BTN_DOWN][f].seek_button[k:0..NUM_PASSENGERS] ->
                if (k>0) then
                        OPENING_DOORS[p][f]
                else
                        CLOSING_DOORS_WA6[p][f]
        )
        else
        //      WA              Test for external calls at the boundary
        //                      floors separately, as the direction has
        //                      not been flipped as yet.
        if ( d == UP && f == TOP_FLOOR ) then
        (
                [BTN_DOWN][f].seek_button[l:0..NUM_PASSENGERS] ->
                if (l>0) then
                (
                        [LIFT_DIR].down ->
                        OPENING_DOORS[p][f]
                )
                else
                        CLOSING_DOORS_WA6[p][f]
        )
        else
        if ( d == DOWN && f == BOTTOM_FLOOR ) then
        (
                [BTN_UP][f].seek_button[m:0..NUM_PASSENGERS] ->
                if (m>0) then
                (
                        [LIFT_DIR].up ->
                        OPENING_DOORS[p][f]
                )
                else
                        CLOSING_DOORS_WA6[p][f]
        )
        else
                CLOSING_DOORS_WA6[p][f]

),


CLOSING_DOORS_WA6[p:OCCUPANTS][f:FLOOR] =
(

        door_is_closed_WA6 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->
        //      WA_NOTE1                Can skip internal walk if lift is empty
        //              (Given *WELL_BEHAVED* passengers)

        if (d==UP) then
                CLOSING_DOORS_LOOK_UP_INTERNAL[p][f][f]
        else
        if (d==DOWN) then
                CLOSING_DOORS_LOOK_DOWN_INTERNAL[p][f][f]

),

CLOSING_DOORS_LOOK_UP_INTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(
```

```
        //      Excerpt from WALK ALGORITHM
        //
        //      6.              test every floor internally in current lift
        //                      direction until boundary floor, starting at the
        //                      current floor (+-1).
        //      7.              if true, IN_TRANSIT in that direction.

        //      NOTE:           We have already tested this floor (f)
        door_is_closed_lui ->
        [LOOKED_UP].check[looked_up:BOOL] ->
        [LOOKED_DOWN].check[looked_down:BOOL] ->

        //              Have we done this part of the walk already,
        //              ie, guard against cycle
        if (    looked_up==TRUE && looked_down==TRUE ) then
        (
                //      14.     If no calls at all then IDLING
                //              But set flags off (ie, reset)
                [LOOKED_UP].no ->
                [LOOKED_DOWN].no ->
                        CLOSING_DOORS_T14[p][f]
        )
        else

        //      Scan floors above this floor (f) internally
        if ( (r+1) <= TOP_FLOOR ) then
        (
                //              Guarded by above check (r+1) <= TOP_FLOOR)
                //              so that we do not continue recursing beyond
                //              boundary floor
                [DPT_COUNT][r+1].seek_button[j:0..MAX_PASSENGERS] ->

                if ( j > 0 && (f+1) <= TOP_FLOOR ) then
                (
                        delay ->
                        [LOOKED_UP].no ->
                        [LOOKED_DOWN].no ->
                        //              There is a call somewhere above (f), so
                        //              travel in that direction.
                        IN_TRANSIT[p][f+1]
                )
                else

                if ( (r+1) < TOP_FLOOR ) then
                        //              Recursive function, will scan every floor
                        //              until boundary is reached
                        CLOSING_DOORS_LOOK_UP_INTERNAL[p][f][r+1]

                else
                        //      8.      else, when boundary reached, test in current
                        //              direction for external calls for current
                        //              direction
                        //              (starting from current floor (+-1)).
                        CLOSING_DOORS_LOOK_UP_EXTERNAL[p][f][f]
        )
        else
                //      8.      else, when boundary reached, test in current
                //              direction for external calls for current direction
                //              (starting from current floor (+-1)).
                CLOSING_DOORS_LOOK_UP_EXTERNAL[p][f][f]

),

CLOSING_DOORS_LOOK_UP_EXTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      8.      else, when boundary reached, test in current
        //              direction for external calls for current direction
        //              (starting from current floor (+-1)).
        //      9.      if any true, IN_TRANSIT in that direction
        //      NOTE:   We have already tested this floor (f)
        door_is_closed_lue ->
```

```
        //              Scan floors above this floor (f) externally
        if ( (r+1) < TOP_FLOOR ) then
        (
                [BTN_UP][r+1].seek_button[j:0..MAX_PASSENGERS] ->

                if ( j > 0 && (f+1) < TOP_FLOOR ) then
                (
                        delay ->
                        [LOOKED_UP].no ->
                        [LOOKED_DOWN].no ->
                        //              There is a call somewhere above (f), so
                        //              travel in that direction.
                        IN_TRANSIT[p][f+1]
                )
                else
                if ( (r+1) < TOP_FLOOR ) then
                        //              Recursive function, will scan every floor
                        //              until boundary is reached
                        CLOSING_DOORS_LOOK_UP_EXTERNAL[p][f][r+1]
                else
                        //      10.     else, when boundary reached, test in current
                        //              direction for external calls for opposite
                        //              direction
                        //              (starting from current floor (+-1).
                        CLOSING_DOORS_LOOK_UP_EXTERNAL_OPPOSITE[p][f][f]
        )
        else
                //      10.     else, when boundary reached, test in current
                //              direction for external calls for opposite direction
                //              (starting from current floor (+-1).
                CLOSING_DOORS_LOOK_UP_EXTERNAL_OPPOSITE[p][f][f]

),

CLOSING_DOORS_LOOK_UP_EXTERNAL_OPPOSITE[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      10.     else, when boundary reached, test in current
        //              direction for external calls for opposite direction
        //              (starting from current floor (+-1).
        //      11.     if any true, IN_TRANSIT in the current direction

        //      NOTE:  We have already tested this floor (f)
        door_is_closed_lue_opp ->

        //      Scan floors above this floor (f) externally
        if ( (r+1) <= TOP_FLOOR ) then
        (
                        [BTN_DOWN][r+1].seek_button[j:0..MAX_PASSENGERS] ->

                        if ( j > 0 && (f+1) <= TOP_FLOOR ) then
                        (
                                delay ->
                                [LOOKED_UP].no ->
                                [LOOKED_DOWN].no ->
                                //      There is a call somewhere above (f), so
                                //      travel in that direction.
                                IN_TRANSIT[p][f+1]
                        )

                        else
                        if ( (r+1) < TOP_FLOOR ) then
                                //      Recursive function, will scan every floor
                                //      until boundary is reached
                                CLOSING_DOORS_LOOK_UP_EXTERNAL_OPPOSITE[p][f][r+1]

                        else
                        (
                                //      12.             else, flip lift direction,
                                //      13.             repeat from 5-12 (lift
                                //                      direction has been flipped
                                [LOOKED_UP].yes ->
                                [LIFT_DIR].down ->
                                CLOSING_DOORS_LOOK_DOWN_INTERNAL[p][f][f]
```

```
                        )
        )
        else
        (
                //      12.     else, flip lift direction,
                //      13.     repeat from 5-12 (lift direction has been flipped
                [LOOKED_UP].yes ->
                [LIFT_DIR].down ->
                CLOSING_DOORS_LOOK_DOWN_INTERNAL[p][f][f]
        )
),


CLOSING_DOORS_LOOK_DOWN_INTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      6.      test every floor internally in current lift
        //              direction until boundary floor, starting at the
        //              current floor (+-1).
        //      7.      if true, IN_TRANSIT in that direction.

        //      NOTE:   We have already tested this floor (f)
        door_is_closed_ldi ->
        [LOOKED_UP].check[looked_up:BOOL] ->
        [LOOKED_DOWN].check[looked_down:BOOL] ->

        //      Have we done this part of the walk already,
        //      ie, guard against cycle
        if (    looked_up==TRUE && looked_down==TRUE ) then
        (
                //      14.     If no calls at all then IDLING
                //              But set flags off (ie, reset)
                [LOOKED_UP].no ->
                [LOOKED_DOWN].no ->
                        CLOSING_DOORS_T14[p][f]
        )
        else

        //      Scan floors below this floor (f) internally
        if ( (r-1) >= BOTTOM_FLOOR ) then
        (
                //      Guarded by above check (r-1) >= BOTTOM_FLOOR)
                //      so that we do not continue recursing beyond
                //      boundary floor
                [DPT_COUNT][r-1].seek_button[j:0..MAX_PASSENGERS] ->

                if ( j > 0 && (f-1) >= BOTTOM_FLOOR ) then
                (
                        delay ->
                        [LOOKED_UP].no ->
                        [LOOKED_DOWN].no ->
                        //      There is a call somewhere above (f), so
                        //      travel in that direction.
                        IN_TRANSIT[p][f-1]
                )
                else

                if ( (r-1) > BOTTOM_FLOOR ) then
                        //      Recursive function, will scan every floor
                        //      until boundary is reached
                        CLOSING_DOORS_LOOK_DOWN_INTERNAL[p][f][r-1]

                else
                        //      8.      else, when boundary reached, test in current
                        //              direction for external calls for current
                        //              direction
                        //              (starting from current floor (+-1)).
                        CLOSING_DOORS_LOOK_DOWN_EXTERNAL[p][f][f]
        )
        else
                        //      8.      else, when boundary reached, test in current
                        //              direction for external calls for current
                        //              direction
                        //              (starting from current floor (+-1)).
```

```
                        CLOSING_DOORS_LOOK_DOWN_EXTERNAL[p][f][f]

),


CLOSING_DOORS_LOOK_DOWN_EXTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //              Excerpt from WALK ALGORITHM
        //
        //      8.      else, when boundary reached, test in current
        //              direction for external calls for current direction
        //              (starting from current floor (+-1)).
        //      9.      if any true, IN_TRANSIT in that direction

        //      NOTE:   We have already tested this floor (f)
        door_is_closed_lde ->

        //              Scan floors below this floor (f) externally
        if ( (r-1) > BOTTOM_FLOOR ) then
        (
                [BTN_DOWN][r-1].seek_button[j:0..MAX_PASSENGERS] ->
                if ( j > 0 && (f-1) > BOTTOM_FLOOR ) then
                (
                        delay ->
                        [LOOKED_UP].no ->
                        [LOOKED_DOWN].no ->
                        //      There is a call somewhere below (f), so
                        //      travel in that direction.
                        IN_TRANSIT[p][f-1]
                )
                else
                if ( (r-1) > BOTTOM_FLOOR ) then
                        //      Recursive function, will scan every floor
                        //      until boundary is reached
                        CLOSING_DOORS_LOOK_DOWN_EXTERNAL[p][f][r-1]
                else
                        //      10.     else, when boundary reached, test in current
                        //              direction for external calls for opposite
                        //              direction
                        //              (starting from current floor (+-1).
                        CLOSING_DOORS_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][f]
        )
        else
                //      10.     else, when boundary reached, test in current
                //              direction for external calls for opposite direction
                //              (starting from current floor (+-1).
                CLOSING_DOORS_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][f]

),

CLOSING_DOORS_LOOK_DOWN_EXTERNAL_OPPOSITE[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      10.             else, when boundary reached, test in current
        //                      direction for external calls for opposite direction
        //                      (starting from current floor (+-1).
        //      11.             if any true, IN_TRANSIT in the current direction

        //      NOTE:           We have already tested this floor (f)
        door_is_closed_lde_opp ->

        //      Scan floors above this floor (f) externally
        if ( (r-1) >= BOTTOM_FLOOR ) then
        (
                        [BTN_UP][r-1].seek_button[j:0..MAX_PASSENGERS] ->

                        if ( j > 0 && (f-1) >= BOTTOM_FLOOR ) then
                        (
                                delay ->
                                [LOOKED_UP].no ->
                                [LOOKED_DOWN].no ->
                                //      There is a call somewhere above (f), so
                                //      travel in that direction.
```

```
                                IN_TRANSIT[p][f-1]
                        )

                        else
                        if ( (r-1) > BOTTOM_FLOOR ) then
                                //      Recursive function, will scan every floor
                                //      until boundary is reached
                                CLOSING_DOORS_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][r-1]

                        else
                        (
                                //      12.     else, flip lift direction,
                                //      13.     repeat from 5-12 (lift direction has
                                //              been flipped
                                [LOOKED_DOWN].yes ->
                                [LIFT_DIR].up ->
                                CLOSING_DOORS_LOOK_UP_INTERNAL[p][f][f]
                        )

        )
        else
        (
                //      12.     else, flip lift direction,
                //      13.     repeat from 5-12 (lift direction has been flipped
                [LOOKED_DOWN].yes ->
                [LIFT_DIR].up ->
                CLOSING_DOORS_LOOK_UP_INTERNAL[p][f][f]
        )



),

CLOSING_DOORS_T14[p:OCCUPANTS][f:FLOOR] =
(
        door_is_closed_T14 ->
        //              First!
        //              We must check for calls at the current floor for the
        //              opposite direction! These are bypassed during the walk.
        //              Note, this is not relevant for boundary floors.

        //              Otherwise we can go to idling.
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        if ( d == UP && f < TOP_FLOOR ) then
        (
                [BTN_DOWN][f].seek_button[i:0..NUM_PASSENGERS] ->
                if (i>0) then
                (
                        [LIFT_DIR].down ->
                        OPENING_DOORS[p][f]
                )
                else
                        IDLING[f]
        )
        else
        if ( d == DOWN && f > BOTTOM_FLOOR ) then
        (
                [BTN_UP][f].seek_button[j:0..NUM_PASSENGERS] ->
                if (j>0) then
                (
                        [LIFT_DIR].up ->
                        OPENING_DOORS[p][f]
                )
                else
                        IDLING[f]
        )
        else


        //      T14     lift is empty
        //              (p==0)
        //              AND no calls to attend
        //              (implied by reaching T14)
        if (p==0) then
                IDLING[f]
```

```
),


IN_TRANSIT[p:OCCUPANTS][f:FLOOR] =
(
        call_received[g:FLOOR] -> IN_TRANSIT[p][f]

        |
        at_next_floor[f] ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T16     someone wants to get out here
        //              Do T16 before T15 because it
        //              a) makes more sense (people get out
        //              before they get in)
        //              b) the initial text description
        //              specifies this ordering

        [DPT_COUNT][f].seek_button[i:0..MAX_PASSENGERS] ->
        if ( i>0 ) then
        (
                at_next_floor_T16 -> OPENING_DOORS[p][f]
        )

        else
        //      T15     there is a waiting passenger at this
        //              floor for current direction
        //              (branch on direction, with guards)
        if ( d == UP && f < TOP_FLOOR ) then
        (
                [BTN_UP][f].seek_button[j:0..NUM_PASSENGERS] ->
                if (j > 0) then
                        OPENING_DOORS[p][f]
                else
                        IN_TRANSIT_T18[p][f]
        )
        else
        if ( d == DOWN && f > BOTTOM_FLOOR ) then
        (
                [BTN_DOWN][f].seek_button[k:0..NUM_PASSENGERS] ->
                if ( k > 0 ) then
                        OPENING_DOORS[p][f]
                else
                        IN_TRANSIT_T18[p][f]
        )
        else
                IN_TRANSIT_T18[p][f]

),



IN_TRANSIT_T18[p:OCCUPANTS][f:FLOOR] =
(
        at_next_floor_T18 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->


        //      T18     no one in lift wants to get out here
        //              (implied by not firing T16)
        //      AND     no waiting passenger on this floor for
        //              current direction
        //              (implied by not firing T15)
        //      AND     there are waiting passengers on floors
        //              in direction of travel
        //      NOTE:   PV, unless extend to check for
        //              pass on floor in direction of travel that
        //              wish to travel in opposite direction to current
        //              lift direction
        //              Must seek! <subcomponent of WALK ALGORITHM>
        //              (WA 8, 9)

        if ( d == UP ) then
                IN_TRANSIT_LOOK_UP_EXTERNAL[p][f][f]
```

```
        else
        if ( d == DOWN ) then
                IN_TRANSIT_LOOK_DOWN_EXTERNAL[p][f][f]

),


IN_TRANSIT_LOOK_UP_EXTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      8.      else, when boundary reached, test in current
        //              direction for external calls for current direction
        //              (starting from current floor (+-1)).
        //      9.      if any true, IN_TRANSIT in that direction

        //      NOTE:   We have already tested this floor (f) for the
        //              current direction only
        at_next_floor_lue ->



        //              Scan floors above this floor (f) externally
        if ( (r+1) < TOP_FLOOR ) then
        (
                [BTN_UP][r+1].seek_button[j:0..MAX_PASSENGERS] ->

                if ( j > 0 && f+1<TOP_FLOOR ) then
                (
                        at_next_floor_WA9 ->

                        //      There is a call somewhere above (f), so
                        //      travel in that direction.
                        IN_TRANSIT[p][f+1]
                )
                else
                if ( (r+1) < TOP_FLOOR ) then
                        //      Recursive function, will scan every floor
                        //      until boundary is reached
                        IN_TRANSIT_LOOK_UP_EXTERNAL[p][f][r+1]
                else
                        IN_TRANSIT_LOOK_UP_EXTERNAL_OPPOSITE[p][f][f]

        )
        else
                IN_TRANSIT_LOOK_UP_EXTERNAL_OPPOSITE[p][f][f]

),

IN_TRANSIT_LOOK_UP_EXTERNAL_OPPOSITE[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      10.     else, when boundary reached, test in current/opposite
        //              direction for external calls for opposite direction
        //              (starting from current floor (+-1).
        //      11.     if any true, IN_TRANSIT in the current direction

        //      NOTE:   We have already tested this floor (f)
        at_next_floor_lue_opp ->

        //              Scan floors above this floor (f) externally
        if ( (r+1) <= TOP_FLOOR ) then
        (
                        [BTN_DOWN][r+1].seek_button[j:0..MAX_PASSENGERS] ->

                        if ( j > 0 && (f+1) <= TOP_FLOOR ) then
                        (
                                delay ->
                                [LIFT_DIR].up ->
                                //      There is a call somewhere above (f), so
                                //      travel in that direction.
                                IN_TRANSIT[p][f+1]
                        )
```

```
                        else
                        if ( (r+1) < TOP_FLOOR ) then
                                //      Recursive function, will scan every floor
                                //      until boundary is reached
                                IN_TRANSIT_LOOK_UP_EXTERNAL_OPPOSITE[p][f][r+1]

                        else
                                //      We only going to look at one direction,
                                //      unlike WALK ALGORITHM, so go straight to
                                //      next transition - T20
                                IN_TRANSIT_T19[p][f]
        )
        else
                IN_TRANSIT_T19[p][f]
),


IN_TRANSIT_LOOK_DOWN_EXTERNAL[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      8.      else, when boundary reached, test in current
        //              direction for external calls for current direction
        //              (starting from current floor (+-1)).
        //      9.      if any true, IN_TRANSIT in that direction

        //      NOTE:   We have already tested this floor (f)
        at_next_floor_lde ->

        //              Scan floors below this floor (f) externally
        if ( (r-1) > BOTTOM_FLOOR ) then
        (

                [BTN_DOWN][r-1].seek_button[j:0..MAX_PASSENGERS] ->

                if ( j > 0 && (f-1) > BOTTOM_FLOOR ) then
                (
                        delay ->

                        //      There is a call somewhere below (f), so
                        //      travel in that direction.
                        IN_TRANSIT[p][f-1]
                )
                else
                if ( (r-1) > BOTTOM_FLOOR ) then
                        //      Recursive function, will scan every floor
                        //      until boundary is reached
                        IN_TRANSIT_LOOK_DOWN_EXTERNAL[p][f][r-1]

                else
                        IN_TRANSIT_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][f]
        )
        else
                IN_TRANSIT_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][f]

),



IN_TRANSIT_LOOK_DOWN_EXTERNAL_OPPOSITE[p:OCCUPANTS][f:FLOOR][r:FLOOR] =
(

        //      Excerpt from WALK ALGORITHM
        //
        //      10.     else, when boundary reached, test in current/opposite
        //              direction for external calls for opposite direction
        //              (starting from current floor (+-1).
        //      11.     if any true, IN_TRANSIT in the current direction

        //      NOTE:   We have already tested this floor (f)
        at_next_floor_lde_opp ->

        //              Scan floors above this floor (f) externally
        if ( (r-1) >= BOTTOM_FLOOR ) then
```

```
        (
                        [BTN_UP][r-1].seek_button[j:0..MAX_PASSENGERS] ->

                        if ( j > 0 && (f-1) >= BOTTOM_FLOOR ) then
                        (
                                delay ->
                                [LIFT_DIR].down ->
                                //      There is a call somewhere below (f), so
                                //      travel in that direction.
                                IN_TRANSIT[p][f-1]
                        )

                        else
                        if ( (r-1) > BOTTOM_FLOOR ) then
                                //      Recursive function, will scan every floor
                                //      until boundary is reached
                                IN_TRANSIT_LOOK_DOWN_EXTERNAL_OPPOSITE[p][f][r-1]

                        else
                                //      We only going to look at one direction
                                //      unlike WALK ALGORITHM, so go straight to
                                //      next transition - T19

                                IN_TRANSIT_T19[p][f]
        )
        else
                IN_TRANSIT_T19[p][f]

),


IN_TRANSIT_T19[p:OCCUPANTS][f:FLOOR] =
(
        at_next_floor_T19 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

        //      T19     !( T15 | T16 | T17 | T18 )
        //              (simplified by not firing T16, T15, T17, or T18)
        //      AND     there is a waiting passenger on this
        //              floor for opposite direction of
        //              current travel
        //              (Seek, with necessary guards)

        if ( d == UP && f > BOTTOM_FLOOR) then
        (
                [BTN_DOWN][f].seek_button[j:0..NUM_PASSENGERS] ->
                if (j > 0) then
                (
                        // we need to flip
                        [LIFT_DIR].down ->
                        OPENING_DOORS[p][f]
                )
                else
                        IN_TRANSIT_T17[p][f]
        )
        else
        if ( d == DOWN && f < TOP_FLOOR ) then
        (
                [BTN_UP][f].seek_button[k:0..NUM_PASSENGERS] ->
                if ( k > 0 ) then
                (
                        // we need to flip
                        [LIFT_DIR].up ->
                        OPENING_DOORS[p][f]
                )
                else
                        IN_TRANSIT_T17[p][f]
        )
        else
                IN_TRANSIT_T17[p][f]
),



IN_TRANSIT_T17[p:OCCUPANTS][f:FLOOR] =
```

```
(
        at_next_floor_T17 ->
        [LIFT_DIR].seek_dir[d:DIRECTION] ->

                //      T17     lift is not empty
                //      AND     no one wants to get out here
                //              (implied by not firing T16)
                //      AND     no waiting passenger on this floor for
                //              current direction
                //              (implied by not firing T15)
                //              (continue travelling in current
                //              direction, unless at boundary floor,
                //              in which case do not fire this trans.)

                if ( p != 0 && d == UP && f < TOP_FLOOR ) then
                (
                        at_next_floor_T17 -> IN_TRANSIT[p][f+1]
                )

                else
                if ( p != 0 && d == DOWN && f > BOTTOM_FLOOR ) then
                (
                        at_next_floor_T17 -> IN_TRANSIT[p][f-1]
                )

                else
                        //      T21     !( T15 | T16 | T17 | T18 | T19 | T20 )
                        //              (implied by not firing T15-T20)
                        //              nothing to do, go to IDLING
                        IDLING[f]
)

\{
choice_T4,
choice_T5,
choice_T9,
choice_T12,
choice_T14,
choice_T17,
choice_T19,

cd_wa6,
delay,

cd_l_u_i,
cd_l_u_e,
cd_l_u_e_opp,

cd_l_d_i,
cd_l_d_e,
cd_l_d_e_opp,

it_l_u_e,
it_l_u_e_opp,

it_l_d_e,
it_l_d_e_opp



}.



//-------------------------------------------------------------------------------
//                              LIFT SYSTEM
//-------------------------------------------------------------------------------


||LIFT_SYSTEM =
                        (
                                LIFT    (       'lift_direction,
                                                'dptCount,
                                                'btnUp,
                                                'btnDown,
                                                'looked_up,
```

```
                                        'looked_down
                                )
                        ||      passenger[p:1..NUM_PASSENGERS]:PASSENGER
                        ||              dptCount[d:FLOOR]:BUTTON
                        ||              btnUp[up:1..NUM_FLOORS-1]:BUTTON
                        ||              btnDown[down:2..NUM_FLOORS]:BUTTON
                        ||              lift_direction:LIFT_DIRECTION
                        ||              looked_up:LOOKED
                        ||              looked_down:LOOKED

)
/{
passenger[p:1..NUM_PASSENGERS].call_at_ground_floor/btnUp[1].on,
passenger[p:1..NUM_PASSENGERS].call_at_top_floor/btnDown[NUM_FLOORS].on,
passenger[p:1..NUM_PASSENGERS].call_at_floor[m:MIDDLE_FLOORS][UP]/btnUp[m].on,
passenger[p:1..NUM_PASSENGERS].call_at_floor[m:MIDDLE_FLOORS][DOWN]/btnDown[m].on,
passenger[p:1..NUM_PASSENGERS].call_at_floor[m:MIDDLE_FLOORS][d:DIRECTION]/call_re
ceived[m],
passenger[p:1..NUM_PASSENGERS].call_at_ground_floor/call_received[1],
passenger[p:1..NUM_PASSENGERS].call_at_top_floor/call_received[NUM_FLOORS],
passenger[p:1..NUM_PASSENGERS].enter_lift[f:FLOOR]/enter_lift[f],
passenger[p:1..NUM_PASSENGERS].press_destination[f:FLOOR]/dptCount[f].on,
passenger[p:1..NUM_PASSENGERS].left_lift[f:FLOOR]/dptCount[f].off,
passenger[p:1..NUM_PASSENGERS].left_lift[f:FLOOR]/passenger_gone[f],
passenger[p:1..NUM_PASSENGERS].entered_lift[f:FLOOR]/passenger_is_in[f],
passenger[p:1..NUM_PASSENGERS].press_destination[f:FLOOR]/press_call[f],
passenger[p:1..NUM_PASSENGERS].destination_reached[f:FLOOR]/destination_reached[f]
}.
```